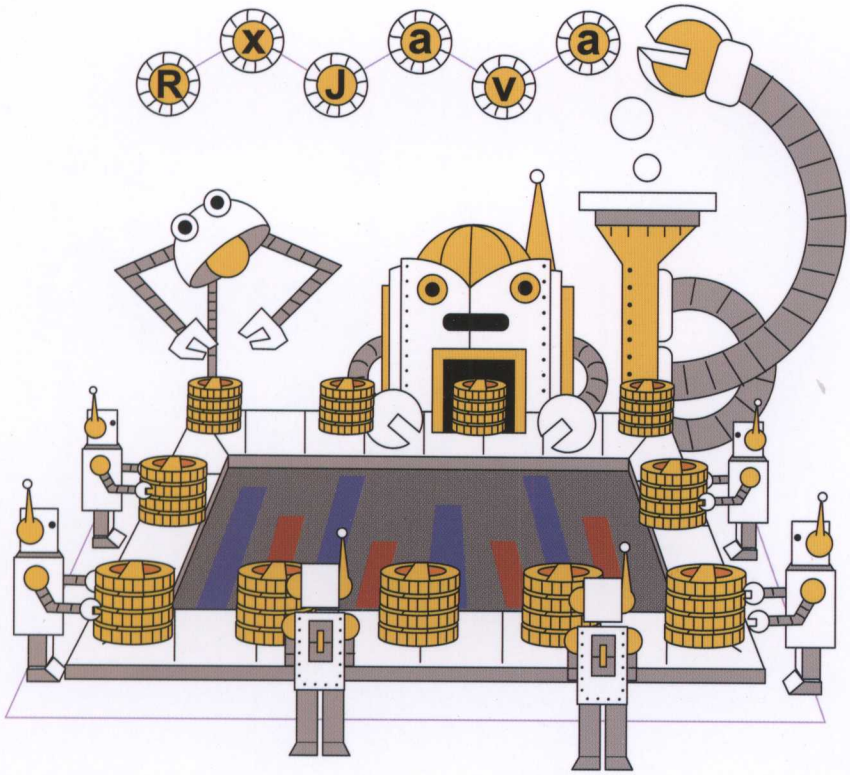
Broadview®
www.broadview.com.cn



RxJava 响应式编程

李衍顺 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

RxJava
响应式编程

李衍顺 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

响应式编程是一种基于异步数据流概念的编程模式。在开发手机 App、Web App 时，要想保证对用户请求的实时响应，给用户带来流畅的体验，响应式编程是一个不错的选择，RxJava 则是这种编程模式的 Java 实现。本书主要介绍如何使用 RxJava 进行响应式编程。全书一共 6 章，从响应式编程与 RxJava 的概念，到 RxJava 的操作符和源码，以及各种 Scheduler 的特点和适用场景，均做了较细致的讲解。本书还用一章的篇幅给出了几个 RxJava 的实用案例，帮助读者理解概念，上手操作。

本书适合 RxJava 的初学者，以及对 RxJava 有初步了解并想要进一步深入学习的读者阅读。

图书在版编目（CIP）数据

RxJava 响应式编程 / 李衍顺著. —北京：电子工业出版社，2018.4
ISBN 978-7-121-33640-9

Ⅰ. ①R… Ⅱ. ①李… Ⅲ. ①移动电话机－应用程序－程序设计 ②JAVA 语言－程序设计

Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2018）第 022880 号

策划编辑：许　艳
责任编辑：张春雨
印　　刷：三河市君旺印务有限公司
装　　订：三河市君旺印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×980　1/16　印张：14.25　字数：299 千字　印数：2500 册
版　　次：2018 年 4 月第 1 版
印　　次：2018 年 4 月第 1 次印刷
定　　价：49.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

前 言

毫无疑问，RxJava 是一个非常优秀的开源库，清晰的流式操作和便捷的线程切换为 Java 和 Android 开发者提供了有力的帮助。网上有大量介绍 RxJava 的文章，开发者可以很容易地查找到相关的学习资料。但是由于 RxJava 入门比较困难，而且缺乏一本系统地介绍 RxJava 的中文书籍，所以给很多初学者带来了困扰，不少人浅尝辄止，放弃了深入学习和使用 RxJava 的机会，这十分可惜。本书作为一本入门书，比较适合 RxJava 的初学者以及对 RxJava 有初步了解并想要进一步学习 RxJava 的读者。

内容结构

本书第 1 章从响应式编程入手，介绍了 RxJava 及 RxJava 的组成部分，帮读者初步了解 RxJava。

第 2 章配合官方的示意图分类介绍了 RxJava 的大部分操作符。这一章的篇幅比较多，读者在阅读的时候可能无法全部记住，可以在需要时随时翻阅查询。

第 3 章就各种 Scheduler 的特点和适合的使用场景做了介绍，帮助读者根据实际需要选择最合适的 Scheduler。

只知道轮子怎么跑还不够，还有必要知道轮子是如何造的，第 4 章结合源码研究了 RxJava 的实现原理。了解原理一方面可以让我们避免用错操作符或者 Scheduler，另一方面

如果碰到 RxJava 中的 bug，也有助于我们定位 bug。发现 bug 后可以到 GitHub 上发起一个 issue，而且最好能够提一个附带的 pull request 来修复这个 bug。

第 5 章给出了一些实例和基于 RxJava 的开源库的使用示例，以帮助读者更好地将 RxJava 应用于实际开发中。

第 6 章介绍了 RxJava 2 相对于 RxJava 1 的改进之处，如果读者已经掌握了 RxJava 1，那么 RxJava 2 也可以很容易地上手。

给初学者的建议

RxJava 这种响应式编程方式跟大多数人习惯的命令式编程方式有较大的区别，所以初学者首先需要完成编程思想上的转变，理解 RxJava 的思想。如可以将 Observable 看作工厂的原材料生产机器，发送出来的数据即为原材料，整个链式操作可以视为原材料经过一条流水线，每个操作符为流水线上的一个车间，每个车间都会对原材料做一定的加工，最终的 Subscriber 可以视为最终消费者，会接收加工后的成品。

其次就是了解 RxJava 的操作符都有哪些，都有什么样的作用。你不需要一开始就将每个操作符都记住，但是可以大体上记住都有什么功能的操作符，这样在需要时就能够想起哪个操作符能够满足当下的需求。关于操作符的详细使用方式可以参阅第 2 章。

接下来就是实践环节了。初期可以尝试应用 RxJava 写一些小程序，并参阅网上的一些开源代码，看看别人都是怎么应用 RxJava 的。初步掌握之后就可以逐渐将 RxJava 引入到项目中，来解决一些工作过程中遇到的实际问题。只看不做永远都是眼高手低，只有将 RxJava 真正地应用到实际开发工作中，不断犯错、不断改进才能真正达到融会贯通的地步，才能真正地掌握 RxJava 的使用技巧。

最后，如果想要进一步学习 RxJava，可以阅读源代码，可以深入地跟踪一个操作符的实现过程来了解其原理。如果有可能，可以参与到 RxJava 的 bug 修复或者新功能开发中，在 GitHub 上给 RxJava 提 pull request，上面有很多大神会给你提各种修改意见，理解他们的思路绝对会让你受益匪浅。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

下载资源：本书如提供示例代码及资源文件，均可在 下载资源 处下载。

提交勘误：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/33640

目 录

第 1 章
走进 RxJava 的世界

1.1 响应式编程

什么是响应式编程？响应式编程是一种以异步数据流为核心的编程方式。这里的数据一般是一些事件；而流则是在时间序列上的一系列的事件。任何东西都可以转化为数据流，如变量、用户输入事件、数据结构等。

我们可以很灵活地操纵数据流，如可以将两个甚至多个数据流融合成一个数据流，可以从数据流中过滤出感兴趣的事件，还可以将数据流中的事件转化为其他新的事件。数据流中的事件通常可以分成三种类型：普通事件、错误事件和结束事件。以用户的键盘输入事件为例，当用户依次敲击“A”、“B”、“C”键的时候，就会产生三个输入事件，计算机接收到这些事件并对其做出响应——将字母“A”、“B”、“C”显示在显示器上。当用户敲击回车键时，可以将其作为一个结束事件来表示数据流的结束，即用户输入结束。而在输入过程中发生的任何错误都可以作为数据流中的错误事件。

我们为什么要使用响应式编程呢？考虑这样的一种场景：用户登录一个购物客户端，这时客户端会将自己的用户名和密码通过网络请求发送出去，然后通过注册一个回调接口来监听请求的结果，因为结果只有成功和失败两种，所以回调里需要有 onSuccess 和 onError 两个接口来分别处理这两种情况，如代码 1-1-1 所示。

代码 1-1-1

```
void sendRequest(String userName,String passwrod) {
    client.sendLoginRequest(userName, passwrod, new Callback() {
```

```
        public void onSuccess(UserInfo info) {
            //处理登录成功的操作
        }
        public void onError(Exception e) {
            //处理错误的操作
        }
    });
}
```

通过注册一个回调接口来监听请求的结果，这其实也算一种响应式编程了。如果在登录成功后，我们要根据用户的 ID 来请求用户的购物记录，又该怎么办呢？很明显，应该在 onSuccess 方法里面继续请求。同登录请求一样，也需要注册一个回调接口来监听请求的结果，让我们继续在代码 1-1-1 的基础上添加代码，如代码 1-1-2 所示。登录成功后我们发出了查询购物记录的请求，并在 onSuccess 方法里将购物记录展示到 UI 上面。

代码 1-1-2

```
client.sendLoginRequest(userName, passwrod, new LoginCallback() {
    public void onSuccess(UserInfo info) {
        client.sendRecordRequest(info.ID, new RecordCallback() {
            public void onSuccess(List<Record> list) {
                //在 UI 上展示购物记录
                view.update(list);
            }
            public void onError(Exception e) {
                //处理错误的操作
            }
        });
    }
    public void onError(Exception e) {
        //处理错误的操作
    }
});
```

到这一步感觉还不错——除了有回调的嵌套外。如果我们有更进一步的需求：要求只展示最近一个月的购物记录（不考虑服务器端的实现），该怎么办呢？看来只能继续在 onSuccess 方法里改了，如代码 1-1-3 所示。为了看起来简单，我们忽略了外层的嵌套。

代码 1-1-3

```
public void onSuccess(List<Record> list) {
    Iterator<Integer> iterator = list.iterator();
    while (iterator.hasNext()) {
        Record record = iterator.next();
        if (record.time < getTime())
            iterator.remove();
    }
    //在 UI 上展示购买记录
}
```

新的需求又来了：每条购买记录里可能包含多个物品，之前只展示了每条购买记录，现在我们需要将所有购买记录里的所有物品都展示到 UI 上。继续改，如代码 1-1-4 所示。

代码 1-1-4

```
public void onSuccess(List<Record> list) {
    Iterator<Integer> iterator = list.iterator();
    while (iterator.hasNext()) {
        Record record = iterator.next();
        if (record.time < getTime())
            iterator.remove();
    }
    List<Item> itemList = new ArrayList<>();
    for (Record record : list) {
        for (Item item : record.getItemList()) {
            itemList.add(item);
        }
    }
    //在 UI 上展示物品记录
    view.update(itemList);
}
```

终于再次改完了，但是还可能有其他需求出现，如只显示金额大于 100 元的物品、将所有相同的物品合并后显示、有些物品需要请求网络获取更详细的信息，等等。看到这里，你会不会很崩溃？传统的命令式编程在处理这样的需求时就是非常痛苦的，但是使用响应式编程就可以游刃有余地处理。如果使用响应式编程，这些要求可以很容易地串成一条链式调用，如图

1-1-1 所示。读者可能不是很理解图中每一步的操作符是什么意思，不要担心，相信你读完本书后就可以很轻松地将该图中的链式调用实现出来。

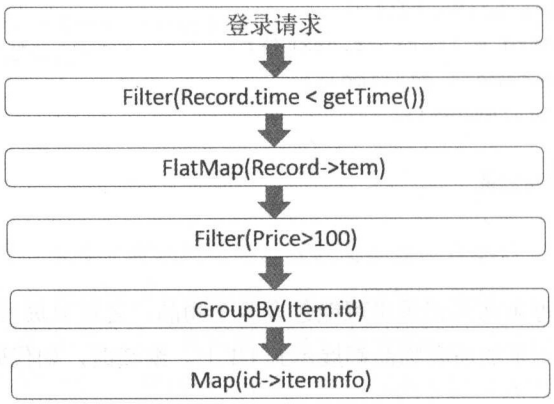


图 1-1-1

通过上面的实例和图 1-1-1，我们可以看到响应式编程提高了代码的抽象层级，所以我们只需要关注与业务逻辑相关的事件，而不必去纠缠里面的一些细节。不仅如此，响应式编程的链式调用还可以消除回调嵌套，所以使用响应式编程写出来的代码往往会更加简明易懂。响应式编程非常适合以下几种情况：

1. 鼠标的点击、移动事件，键盘的输入事件，移动设备上的各种触摸和手势事件。
2. 当用户的位置改变时，其移动设备上的 GPS 信号和陀螺仪信号。
3. 各种耗时的操作，如读取硬盘内容以及从网络请求数据，这些操作一般都是异步的。
4. 涉及数据的转化、组合、过滤等操作的场景。

1.2 什么是 RxJava

RxJava 是一个非常著名的开源库，是 ReactiveX（Reactive Extensions）的一种 Java 实现。ReactiveX 是一种响应式扩展框架，有很多种实现，如 RxAndroid、RxJS、RxSwift、RxRuby、RxCpp、RxGo 等。最近几年 RxJava 在 Java 和 Android 的开发中得到了广泛的使用。截至本书写作时，RxJava 在 GitHub 上已经超过 28000 个 Star，想来突破 30000 也不是很遥远的事情。目

前 RxJava 有 1.x 和 2.x 两个主要的分支，分别代表着 RxJava 1 和 RxJava 2。由于 RxJava 1 发布的时间较早，使用也更广泛，所以本书的内容主要是针对 RxJava 1 的。但是 RxJava 2 已经发布，以后也会逐渐流行起来，所以在本书的最后一章会对 RxJava 2 做一些简单的介绍。

RxJava 可以看作由 Observable、Subscriber 和 Scheduler 组成的，它们的关系如图 1-2-1 所示。Subscriber 订阅到 Observable，Observable 会在默认或者指定的 Scheduler 上工作并产生数据流返回给 Subscriber，Subscriber 也会在默认或者指定的 Scheduler 上接收 Observable 发送过来的数据流。Scheduler 是对线程的一种抽象，不同的 Scheduler 代表了不同的线程。

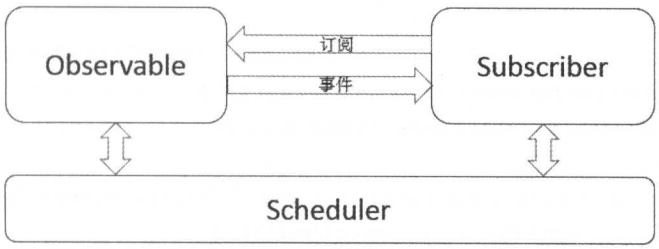


图 1-2-1

1.3　Observable 和 Subscriber

Observable 和 Subscriber 是 RxJava 的重要组成部分。Observable 提供了 subscribe 方法，当有 Subscriber 通过 subscribe 方法订阅到 Observable 时，Observable 就可以向 Subscriber 发送数据流。在 1.1 节中我们了解到响应式编程中的事件分为三类：普通事件、错误事件和结束事件，在 Subscriber 中有三个方法与这三种事件一一对应，Observable 会通过调用 Subscriber 的这三个方法来发送对应的事件。

1. onNext：当 Observable 要发送普通事件时，就会调用这个方法。这个方法可以被调用 0～*N* 次。

2. onError：当在 Observable 内部有异常或者错误产生的时候，就可以调用这个方法来向 Subscriber 发送错误事件。这个方法只能被调用 1 次。

3. onComplete：如果 Observable 已经发送完所有的数据，并且没有发生错误，这时就需要调用这个方法来向 Subscriber 发送结束事件。这个方法也只能调用 1 次，而且和 onError 是

互斥的关系，也就是说调用了 onError 后就不能调用 onComplete，反之亦然。在 onError 或者 onComplete 被调用之后，Observable 就失去了作用，不能再调用 onNext 来发送数据了。

Subscriber 还提供了 unsubscribe 方法，当 Subscriber 订阅到 Observable 之后，可以随时调用这个方法来终止对 Observable 的订阅。

代码 1-3-1 定义了一个方法，该方法创建一个 Observable 并将其返回。创建的 Observable 会发送随机生成的 5 个小于 10 的整数。当生成的随机数大于 8 时，我们假设有了异常，然后调用 onError 抛出一个异常。

代码 1-3-1

```
private Observable<Integer> createObserver() {
    return Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            if (!subscriber.isUnsubscribed()) {
                for (int i = 0; i < 5; i++) {
                    int temp = new Random().nextInt(10);
                    if (temp > 8) {
                        //如果 value>8，则创建一个异常
                        subscriber.onError(new Throwable("value >8"));
                        break;
                    } else {
                        subscriber.onNext(temp);
                    }
                    // 没有发生异常，正常结束
                    if (i == 4) {
                        subscriber.onCompleted();
                    }
                }
            }
        }
    });
}
```

接下来建立一个 Subscriber 对象并将其注册给创建的 Observable 对象，然后接收其发送来的数据，参见代码 1-3-2。其中 log 函数是自定义的函数，会将结果输出。

代码 1-3-2

```
createObserver().subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        log("onComplete!");
    }
    @Override
    public void onError(Throwable e) {
        log("onError:" + e.getMessage());
    }
    @Override
    public void onNext(Integer integer) {
        log("onNext:" + integer);
    }
});
```

多测试几次会产生不同的结果，例如下面是测试两次的结果，第一次顺利发送完了 5 个数据，第二次在发送了 3 个数据后产生了错误。

```
onNext:3
onNext:8
onNext:2
onNext:8
onNext:8
onComplete!
onNext:0
onNext:1
onNext:6
onError:value >8
```

除了上面这种通用的 Observable 之外，还有两种特殊的 Observable，同样也可以产生并发送数据，它们分别是 Single 和 Completable，与之分别对应的 Subscriber 是 SingleSubscriber 和 CompletableSubscriber。

1.3.1 Single：单个数据的生产者

Observable 产生的是一系列的多个数据，但是在很多情况下我们只需要一个数据，如使用 Get 方法从网络接口读取数据等，在这种情况下就可以使用 Single 来代替 Observable 进行数据的发送。Single 同 Observable 类似，能够产生并发送数据和事件。但是不同于 Observable 可以发送无穷多的数据，Single 只能发送一个数据或者一个错误事件。Single 虽然也可以接受 Subscriber 的订阅，但是基于 Single 的特点，一般使用 SingleSubscriber 订阅。Single 会调用 SingleSubscriber 的两个方法来发送数据和事件。

1. onSuccess：发送 Single 唯一的数据，只能调用一次。

2. onError：Single 内部有异常或者错误时，调用此接口发送异常事件，同样只能调用一次。

同 Observable 一样，Single 可以使用非常丰富的操作符（在第 2 章中会详细介绍各种操作符），其对大部分操作符的使用方式也和 Observable 类似。有些操作符可以让 Single 和 Observable 互相转化。代码 1-3-3 创建了一个 Single 对象，并对其进行订阅，订阅后会输出结果“1”。

代码 1-3-3

```
Single.create(new Single.OnSubscribe<Integer>() {
    @Override
    public void call(SingleSubscriber<? super Integer>singleSubscriber) {
        if (!singleSubscriber.isUnsubscribed()) {
            singleSubscriber.onSuccess(1);
        }
    }
}).subscribe(new SingleSubscriber<Integer>() {
    @Override
    public void onSuccess(Integer value) {
        log(value);
    }
    @Override
    public void onError(Throwable error) {
        log(error.getMessage());
    }
});
```

代码 1-3-3 只发送了一个数据，代码 1-3-4 不会发送数据，而是会发送一个错误事件，订阅后就会输出结果“Single error”。

代码 1-3-4

```
Single.create(new Single.OnSubscribe<Integer>() {
    @Override
    public void call(SingleSubscriber<? super Integer>singleSubscriber) {
        if (!singleSubscriber.isUnsubscribed()) {
            singleSubscriber.onError(new Throwable("Single error"));
        }
    }
}).subscribe(new SingleSubscriber<Integer>() {
    @Override
    public void onSuccess(Integer value) {
        log(value);
    }
    @Override
    public void onError(Throwable error) {
        log(error.getMessage());
    }
});
```

1.3.2　Completable：单个事件的生产者

通过 1.3.1 节我们了解了 Single，它能够产生并发送数据。但是在有些情况下我们不需要数据，只需要关心结果是成功还是失败，如使用 Put 方法将数据上传到网络服务器，我们只关心上传是否成功，而不关心数据本身。在这种情况下我们就可以使用 Completable。Completable 与 Single 的不同之处在于，其只会发送错误和结束的事件，而不发送数据。Completable 通过调用 CompletableSubscriber 的以下三个方法来发送事件。

1. onComplete：在 Completable 的工作顺利完成后，就可以调用此方法来发送结束事件。

2. onError：当 Completable 内部有异常或者错误产生时，调用此方法来发送错误事件。

3. onSubscribe(Subscription d)：Completable 可以调用此接口来返回一个 Subscription 对象，通过此对象可以取消对 Completable 的订阅。

与 Observable 类似，Completable 同样有很多的操作符可以使用，也有一些同 Observable 和 Single 进行转化的操作符。下面的代码创建了两个 Completable 对象并分别对其进行订阅，代码 1-3-5 会输出“Completable error”，而代码 1-3-6 则会输出“onCompleted”。

代码 1-3-5

```
Completable.error(new Throwable("Completable error"))
        .subscribe(new CompletableSubscriber() {
            @Override
            public void onCompleted() {
                log("onCompleted");
            }
            @Override
            public void onError(Throwable e) {
                log(e.getMessage());
            }
            @Override
            public void onSubscribe(Subscription d) {
            }
        });
```

代码 1-3-6

```
Completable.complete()
        .subscribe(new CompletableSubscriber() {
            @Override
            public void onCompleted() {
                log("onCompleted");
            }
            @Override
            public void onError(Throwable e) {
                log(e.getMessage());
            }
            @Override
            public void onSubscribe(Subscription d) {
            }
        });
```

1.4　在 Android 工程中引入 RxJava

RxJava 现在被广泛应用到了 Android 和 Java 开发中，如果想在 Andorid 工程中使用 RxJava 的话，只需要在 Gradle 配置文件中加入对 RxJava 的依赖。笔者写这本书的时候，RxJava 的最新版本是 1.3.3，因此可以通过下面的配置来加入对 RxJava 1.3.3 的依赖。

```
compile 'io.reactivex:rxjava:1.3.3'
```

但是在 Android 开发中一般会添加对 RxAndroid 的依赖，而 RxAndroid 已经依赖于 RxJava，并且一般难以及时更新到最新版的 RxJava，所以如果要使用 RxAndroid 和最新版的 RxJava，则可以通过下面的配置来导入依赖。

```
compile('io.reactivex:rxandroid:1.2.1') {
    exclude module: 'rxjava'
}
compile 'io.reactivex:rxjava:1.3.3'
```

由于 RxJava 2 早已发布，你可能想直接用 RxJava 2，这时可以通过下面的配置来导入对 RxJava 2 的依赖，其中 x 和 y 可根据 RxJava 2 的发布情况来随时更新，以获取最新版本的 RxJava 2。

```
compile "io.reactivex.rxjava2:rxjava:2.x.y"
```

第 2 章
RxJava 中的操作符

丰富的操作符是 RxJava 功能强大的体现之一。所谓的操作符其实就是一系列的函数，这些函数接收 Observable<T>类型的输入，生成 Observable<R>类型的输出，这里 T 和 R 可以相同也可以不相同。因为输入和输出都是 Observable 类型的，所以多个操作符可以共同形成链式调用，让程序的逻辑操作非常简明。

通常情况下，只要业务稍微复杂点，Observable 产生的数据就不能直接满足 Subscriber 的需求。如从网络请求的数据可能是 json 格式的原始数据，在程序中不能直接拿来用，而需要将其转化成对应的 Bean，然后发送到 Subscriber，这样就可以直接用来做更新 UI 等操作。所以要想灵活地运用 RxJava 来处理各种各样的业务需求，熟练掌握操作符是十分有必要的。接下来将按照操作符的不同功能分类来讲解一下常用的操作符。[1]

2.1 创建 Observable 的操作符

其实在第 1 章中我们已经用到了一种创建 Observable 的操作符了，那就是 create。在平时创建 Observable 时不推荐直接使用 create 操作符，推荐优先使用其他的操作符。这是因为使用 create 创建的 Observable 可能会因为不满足 RxJava 的规范而造成一些潜在的不稳定因素，比如

1：本书中所有的示意图都是 RxJava 社区的原图，版权属于 RxJava 社区。

发送数据和事件前没检查 Subscriber 的状态，调用 onError 之后又调用了 onComplete，工作完成后忘记调用 onComplete 等。接下来我们来看一下其他用于创建 Observable 的操作符。

2.1.1　range

顾名思义，range 操作符创建的 Observable 将会发送一个范围内的数据，其示意图如图 2-1-1 所示。

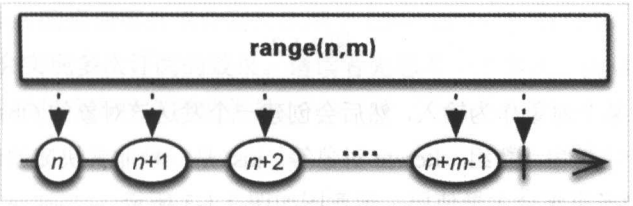


图 2-1-1

从示意图中可以看到，range 操作符接收两个参数 n 和 m，创建的 Observable 将依次发送从 *n* 开始的 *m* 个数。需要注意的是，最后一个数为 *n*+*m*−1，而不是 *n*+*m*。代码 2-1-1 使用 range 操作符创建了一个 Observable，对其订阅后就会输出 10~14 的 5 个整数。

代码 2-1-1

```
Observable.range(10, 5).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        log(integer);
    }
});
```

2.1.2　defer 和 just

defer 操作符的示意图如图 2-1-2 所示，只有当有 Subscriber 来订阅的时候才会创建一个新的 Observable 对象，也就是说每次订阅都会得到一个刚创建的最新的 Observable 对象，这可以确保 Observable 对象里的数据是最新的。其特点将在后面和 just 操作符对比讲解。

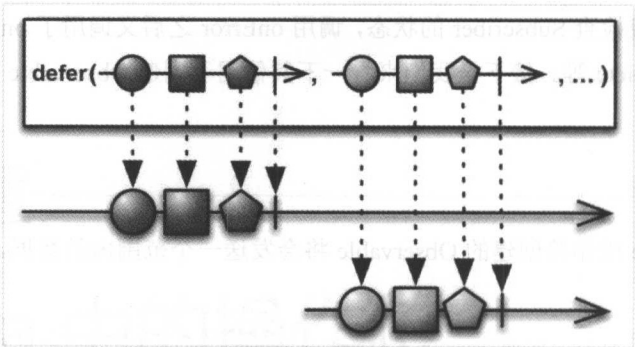


图 2-1-2

just 操作符接收某个对象作为输入，然后会创建一个发送该对象的 Observable。这个对象可以是一个数字、一个字符串、数组、Iterate 对象等。just 是一种非常快捷的创建 Observable 对象的方法，在后面的例子里将会大量使用，示意图如图 2-1-3 所示。

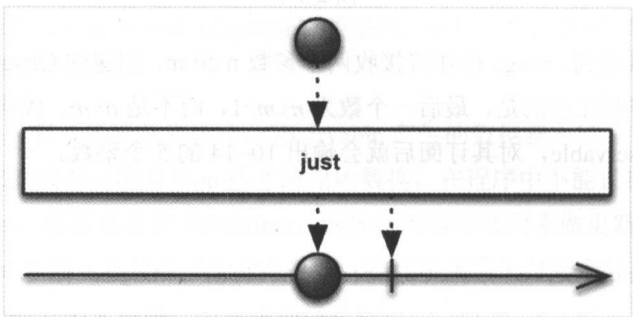


图 2-1-3

代码 2-1-2 通过两个方法创建两个 Observable，这两个 Observable 都会将 Android 系统中的当前时间作为数据发送。我们将这两个 Observable 分别保存在变量 deferObservable 和 justObservable 中。

代码 2-1-2

```
Observable<Long> deferObservable = getDefer();
Observable<Long> justObservable = getJust();
private Observable<Long> getJust() {
    return Observable.just(System.currentTimeMillis());
}
```

```
private Observable<Long> getDefer() {
    return Observable.defer(new Func0<Observable<Long>>() {
        @Override
        public Observable<Long> call() {
            return getJust();
        }
    });
}
```

现在在两个 Button 的点击事件里分别对这两个 Observable 进行订阅，如代码 2-1-3 所示，多次点击 Button 就会实现多次订阅的效果。请读者先思考一下多次订阅后 deferObservable 和 justObservable 发送的数据会有什么样的差异。

代码 2-1-3

```
deferObservable.subscribe(new Action1<Long>() {
    @Override
    public void call(Long time) {
        log("defer:" + time);
    }
});
justObservable.subscribe(new Action1<Long>() {
    @Override
    public void call(Long time) {
        log("just:" + time);
    }
});
```

输出结果如下所示，可以看到 defer 每次订阅都会得到 Observable 发送的一个全新的当前时间，而 just 创建的操作符即使订阅多次也都会发送出和首次订阅一样的数据。

```
defer:1501895934904
defer:1501895938894
defer:1501895939768
just:1501895933877
just:1501895933877
just:1501895933877
```

2.1.3 from

from 操作符接收一个对象作为参数来创建 Observable，这个参数对象可以是 Iterable、Callable、Future 和数组等，其示意图如图 2-1-4 所示。from 操作符创建的 Observable 将发送参数对象里的数据，其创建方式类似于 just 操作符，但是 just 操作符创建的 Observable 会将整个参数对象作为数据一下子发送出去。比如说参数对象是一个含有 10 个数字的数组，使用 from 创建的 Observable 就会发送 10 次，每次发送一个数字，而使用 just 创建的 Observable 会一次就将整个数组发送出去。

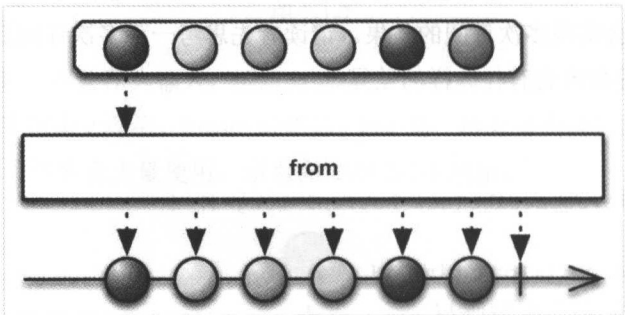


图 2-1-4

代码 2-1-4 首先创建了一个数组和一个 List，分别存储 0~5 的整数，然后使用 from 操作符分别以数组和 List 作为输入参数创建两个 Observable 对象。之后分别对这两个 Observable 对象进行订阅，会输出什么结果呢？

代码 2-1-4

```
Integer[] arrays = {0, 1, 2, 3, 4, 5};
List<Integer> list = new ArrayList<>();
for (int i = 0; i <= 5; i++) {
    list.add(i);
}

private Observable<Integer> FromArray() {
    return Observable.from(arrays);
}

private Observable<Integer> FromIterable() {
```

```
        return Observable.from(list);
    }

    FromArray().subscribe(new Action1<Integer>() {
        @Override
        public void call(Integer i) {
            log("FromArray:" + i);
        }
    });

    FromIterable().subscribe(new Action1<Integer>() {
        @Override
        public void call(Integer i) {
            log("FromIterable:" + i);
        }
    });
```

程序运行后的输出结果如下所示，通过 from 操作符创建的 Observable 对象将数组和 List 中的整数依次发送了出来。

```
FromArray:0
FromArray:1
FromArray:2
FromArray:3
FromArray:4
FromArray:5
FromIterable:0
FromIterable:1
FromIterable:2
FromIterable:3
FromIterable:4
FromIterable:5
```

2.1.4　interval

interval 所创建的 Observable 对象会从 0 开始，每隔固定的时间发送一个数字。需要注意的是，这个对象是运行在 computation Scheduler（Scheduler 将会在第 3 章中进行讲解）中的，所

以在 Android 开发中，如果需要在 UI 中显示结果，则要在主线程中订阅。interval 操作符的示意图如图 2-1-5 所示。

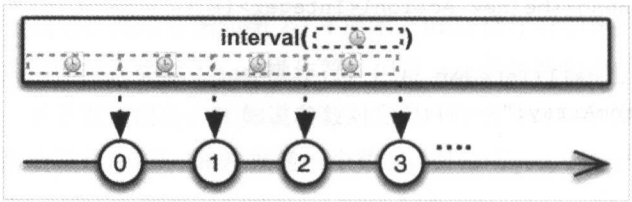


图 2-1-5

代码 2-1-5 通过 interval 操作符创建了一个 Observable，创建的 Observable 将会以 1 秒为间隔不断地发送数据，因为我们需要更新 UI，所以需要在主线程中进行订阅。这里的 AndroidSchedulers.mainThread()属于 RxAndroid 库，RxAndroid 是 Jake Wharton 在 Android 平台上开发的一个对 RxJava 的扩展。最后创建一个 Subscriber 对象对这个 Observable 对象进行订阅，就会每秒输出一个从 0 开始递增的数据。

代码 2-1-5

```
private Observable<Long> interval() {
    return Observable.interval(1, TimeUnit.SECONDS)
    .observeOn(AndroidSchedulers.mainThread());
}
Observable<Long> observable = interval();
Subscriber<Long> subscriber = new Subscriber<Long>() {
    @Override
    public void onCompleted() {
        log("onCompleted" );
    }
    @Override
    public void onError(Throwable e) {
        log("onError:" + e.getMessage());
    }
    @Override
    public void onNext(Long aLong) {
        log(aLong);
    }
};
```

由于 interval 操作符创建的 Observable 对象会不停地发送数据，所以当我们不再需要它的数据时不要忘记调用 unSubscribe 方法进行反订阅，反订阅后 Observable 将会停止发送数据。

```
Subscriber. Unsubscribe
```

2.1.5　repeat 和 timer

repeat 操作符可以让 Observable 对象发送的数据重复发送 *N* 次，我们可以指定其发送的次数，其示意图如图 2-1-6 所示。

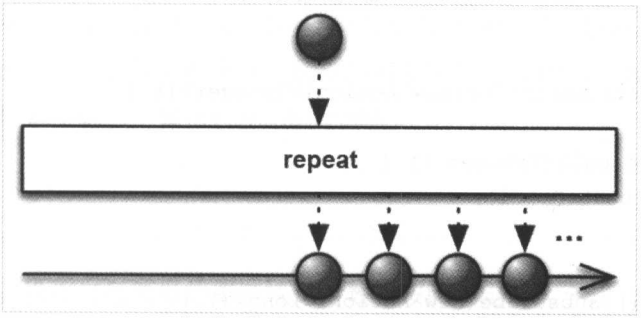


图 2-1-6

timer 操作符创建的 Observable 会在指定时间后发送一个数字 0，注意其默认也是运行在 computation Scheduler 上，其示意图如图 2-1-7 所示。

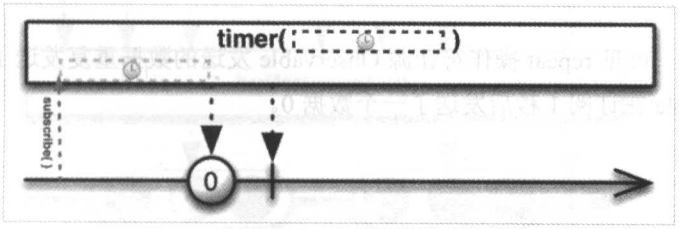


图 2-1-7

下面我们来创建一个发送整数 1、2、3 的源 Observable，并使用 repeat 操作符在源 Observable 的基础上创建一个新的 Observable，使其重复发送数据 3 次。另外使用 timer 创建一个会在 1 秒后发送数据的 Observable，如代码 2-1-6 所示。

代码 2-1-6

```
private Observable<Integer> repeatObserver() {
    return Observable.just(1,2,3).repeat(3);
}
private Observable<Long> timerObserver() {
    //timer 的默认 Scheduler 是 computation Scheduler
    return Observable.timer(1, TimeUnit.SECONDS)
            .observeOn(AndroidSchedulers.mainThread());
}
```

接下来，分别对这两个 Observable 进行订阅，如代码 2-1-7 所示。

代码 2-1-7

```
repeatObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("repeat:" + i);
    }
});
timerObserver().subscribe(new Action1<Long>() {
    @Override
    public void call(Long aLong) {
        log("timer:" + aLong);
    }
});
```

结果如下所示，可见 repeat 操作符让源 Observable 发送的数据重复发送了 3 遍；timer 操作符创建的 Observable 在订阅 1 秒后发送了一个数据 0。

```
repeat:1
repeat:2
repeat:3
repeat:1
repeat:2
repeat:3
repeat:1
repeat:2
```

```
repeat:3
timer:0
```

至此创建 Observable 的操作符就基本讲完了，还有几个非常简单的创建操作符，如 naver、empty、throw 等，在这里就不赘述，感兴趣的读者可以查看官方文档。

2.2　转化 Observable 的操作符

在 2.1 节中，我们了解了如何创建 Observable。如果说创建 Observable 是运用 RxJava 的基础的话，那么转化 Observable 则是其主干。在通常情况下，创建一个 Observable 只是完成了工作的第一步，要想完成一些复杂的工作场景，往往还需要将创建的 Observable 按照一定规则进行转化。在本节中我们来了解一下如何转化 Observable。

2.2.1　buffer

顾名思义，buffer 操作符所要做的事情就是将数据按照规定的大小做一下缓存，当缓存的数据量达到设置的上限后就将缓存的数据作为一个集合发送出去。如示意图 2-2-1 所示，里面的 buffer 的大小为 3，因此每缓存 3 个数据就会将这 3 个数据作为整体发送出去。

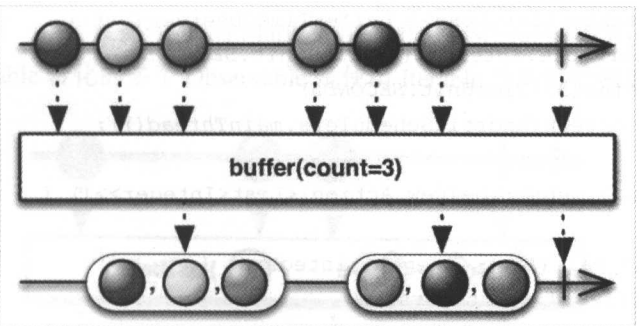


图 2-2-1

buffer 操作符还可以设置跳过的数目。如图 2-2-2 所示，我们加入了一个 skip 参数，用来指定每次发送一个集合需要跳过几个数据。图中指定的 count 为 2，skip 为 3，这样就会每隔 3 个数据发送一个包含 2 个数据的集合。如果 count 大于或者等于 skip 的话，skip 就失去了效果从

而等效于图 2-2-1 中的情况了。

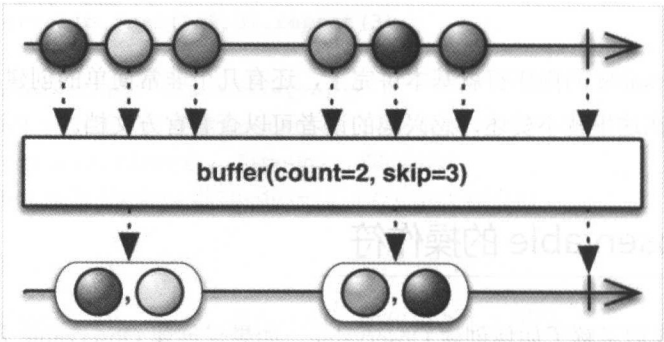


图 2-2-2

buffer 操作符不仅可以通过数量规则来缓存，还可以通过时间等规则来缓存，如规定每 3 秒缓存发送一次，等等。见代码 2-2-1，我们创建了两个 Observable，并使用 buffer 对其进行转化，第一个通过数量规则来缓存，第二个通过时间规则来缓存，然后分别对其进行订阅。

代码 2-2-1

```
private Observable<List<Integer>> bufferObserver() {
    return Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9)
            .buffer(2, 3);
}
private Observable<List<Long>> bufferTimeObserver() {
    return Observable.interval(1, TimeUnit.SECONDS)
            .buffer(3, TimeUnit.SECONDS)
            .observeOn(AndroidSchedulers.mainThread());
}
bufferObserver().subscribe(new Action1<List<Integer>>() {
    @Override
    public void call(List<Integer> integers) {
        log("buffer:" + integers);
    }
});
bufferTimeObserver().subscribe(new Action1<List<Long>>() {
    @Override
    public void call(List<Long> longs) {
        log("bufferTime:" + longs);
```

```
        }
    });
```

运行结果如下所示。可以看到第一个 Observable 会每隔 3 个数字就发送出前 2 个数字；第二个 Observable 会每隔 3 秒输出 2~4 个数字。

```
buffer:[1, 2]
buffer:[4, 5]
buffer:[7, 8]
bufferTime:[0, 1, 2]
bufferTime:[3, 4]
bufferTime:[5, 6, 7]
bufferTime:[8, 9, 10]
```

2.2.2　flatMap

flatMap 是一个用处非常多的操作符，其示意图如图 2-2-3 所示。可以将数据根据我们想要的规则进行转化后再发送出去。其原理就是将这个 Observable 转化为多个以源 Observable 发送的数据作为源数据的 Observable，然后将这多个 Observable 发送的数据整合并发送出来。需要注意的是，数据最后的顺序可能会有交错，如果对顺序有严格要求的话，可以使用 concatmap 操作符。flatMap 还有一个扩展操作符 flatMapIterable，flatMapIterable 和 flatMap 基本相同，不同之处为 flatMapIterable 转化的多个 Observable 是使用 Iterable 作为源数据的。

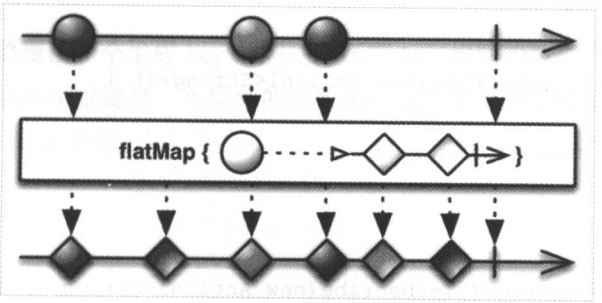


图 2-2-3

下面我们创建 2 个会发送 3 个数据的 Observable，并分别使用 flatMap 和 flatMapIterable 来转化这两个 Observable，转化完成后进行订阅，如代码 2-2-2 和代码 2-2-3 所示。

代码 2-2-2

```
private Observable<String> flatMapObserver() {
    return Observable.just(1, 2, 3)
            .flatMap(new Func1<Integer, Observable<String>>() {
                @Override
                public Observable<String> call(Integer integer) {
                    return Observable.just("flat map:" + integer);
                }
            });
}
private Observable<String> flatMapIterableObserver() {
    return Observable.just(1, 2, 3)
            .flatMapIterable(new Func1<Integer, Iterable<String>>() {
                @Override
                public Iterable<String> call(Integer integer) {
                    ArrayList<String> s = new ArrayList<>();
                    for (int i = 0; i < 3; i++) {
                        s.add("flatMapIterable:" + integer);
                    }
                    return s;
                }
            });
}
```

代码 2-2-3

```
flatMapObserver().subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
        log(s);
    }
});
flatMapIterableObserver().subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
        log(s);
```

```
        }
    });
```

运行后的结果如下所示，flatMap 操作符将源 Observable 发送的数据都转化成字符串并加上了“flat map”的前缀；而 flatMapIterable 不仅会将源 Observable 发送的数据转化成加上了“flatMapIterable”前缀的字符串，而且还会将每个数字扩展为 3 个字符串。

```
flat map:1
flat map:2
flat map:3
flatMapIterable:1
flatMapIterable:1
flatMapIterable:1
flatMapIterable:2
flatMapIterable:2
flatMapIterable:2
flatMapIterable:3
flatMapIterable:3
flatMapIterable:3
```

2.2.3　groupBy

groupBy 操作符会将源 Observable 发送的数据按照 key 来拆分成一些小的 Observable，然后这些小的 Observable 分别发送其所包含的数据，类似于 SQL 里面的 groupBy。

在使用时，我们需要提供一个生成 key 的规则，所有 key 相同的数据会包含在同一个小的 Observable 中。另外我们还可以提供一个额外的函数来对这些数据进行转化，这有点像集成了 flatMap，其示意图如图 2-2-4 所示。

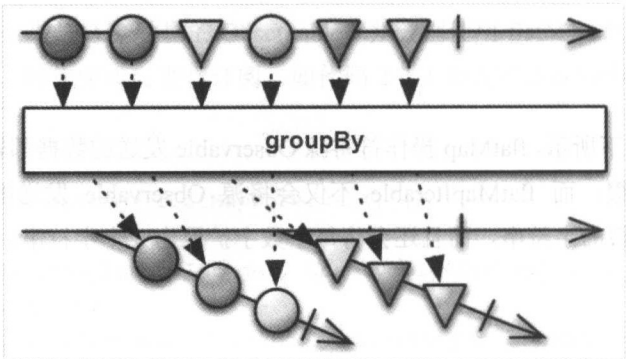


图 2-2-4

代码 2-2-4 将创建两个经过 groupBy 转化的 Observable 对象，第一个按照奇偶分组，第二个添加额外的函数，以实现分组后为数字加上一个字符串前缀。

代码 2-2-4

```
private Observable<GroupedObservable<Integer, Integer>> groupByObserver() {
    return Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9)
            .groupBy(new Func1<Integer, Integer>() {
                @Override
                public Integer call(Integer integer) {
                    return integer % 2;
                }
            });
}
private Observable<GroupedObservable<Integer, String>> groupByStringObserver()
{
    return Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9)
            .groupBy(new Func1<Integer, Integer>() {
                @Override
                public Integer call(Integer integer) {
                    return integer % 2;
                } .
            }, new Func1<Integer, String>() {
                @Override
                public String call(Integer integer) {
                    return "groupByKeyValue:" + integer;
                }
```

```
            });
    }
    groupByObserver().subscribe(new Action1<GroupedObservable<Integer, Integer>>()
{
        @Override
        public void call(GroupedObservable<Integer, Integer> groupedObservable) {
            groupedObservable.count().subscribe(new Action1<Integer>() {
                @Override
                public void call(Integer integer) {
                    log("key" + groupedObservable.getKey()
                            + " contains:" + integer + " numbers");
                }
            });
        }
    });
    groupByStringObserver().subscribe(new Action1<GroupedObservable<Integer,
String>>() {
        @Override
        public void call(GroupedObservable<Integer, String> groupedObservable) {
            if (groupedObservable.getKey() == 0) {
                groupedObservable.subscribe(new Action1<String>() {
                    @Override
                    public void call(String s) {
                        log(s);
                    }
                });
            }
        }
    });
```

需要注意的是，源 Observable 经过 groupBy 转化后发送出来的每个数据是一种特殊的 Observable：GroupedObservable。GroupedObservable 是 Observable 的一个子类，有一个自己的方法 getKey()，可以返回当前 GroupedObservable 的 key。在前一个订阅中，我们将每个 GroupedObservable 通过 count 操作符计算出其大小，然后将 key 和大小一起输出；在后一个订阅中，我们为所有的数据都加上了字符串前缀，并将 key 为 0 的 GroupedObservable 的所有数据都输出，输出的结果如下所示。

```
key0 contains:4 numbers
key1 contains:5 numbers
groupByKeyValue:2
groupByKeyValue:4
groupByKeyValue:6
groupByKeyValue:8
```

2.2.4 map

map 操作符将源 Observable 发送的每个数据都按照给定的函数进行转化，并将转化后的数据发送出来，其示意图如图 2-2-5 所示。map 操作符的功能类似于 flatMap，不同之处在于 map 会对数据直接进行函数转化，输入和输出是一对一的关系；而 flatMap 则将每个数据作为输入来创建一个新的 Observable，并将所有的 Observable 再组合起来发送数据，输入和输出是一对多的关系。所以我们在需要进行数据转化的时候应优先考虑使用 map 操作符，这样可以得到更好的性能。

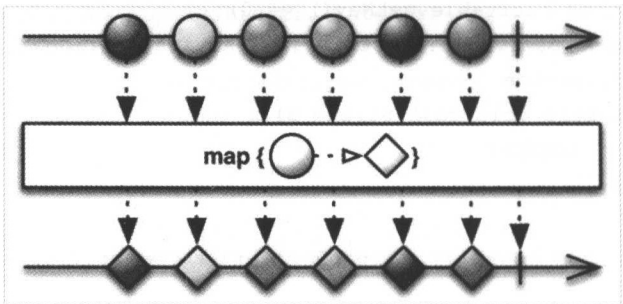


图 2-2-5

代码 2-2-5 首先创建一个 Observable，它能发送 1~3 的整数，然后使用 map 操作符将所有的数据都转化为乘以 10 之后的结果。

代码 2-2-5

```
private Observable<Integer> mapObserver() {
    return Observable.just(1, 2, 3).map(new Func1<Integer, Integer>() {
            @Override
            public Integer call(Integer integer) {
```

```
                return integer * 10;
            }
        });
}
mapObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        log("map:" + integer);
    }
});
```

订阅后输出的结果如下：

```
map:10
map:20
map:30
```

2.2.5　cast

cast 操作符将 Observable 发送的数据强制转化为另外一种类型，和 Java Class 类的 cast 方法很相似，属于 map 的一种具体的实现，其示意图如图 2-2-6 所示。

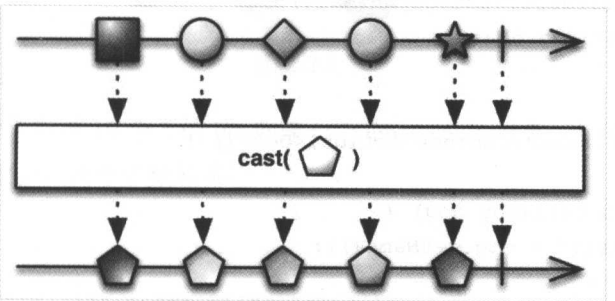


图 2-2-6

要试验一下 cast 操作符的用处，我们先创建一个 Animal 类，然后创建一个继承自 Animal 类的 Dog 类，最后通过一个返回类型为 Animal 的方法 getAnimal 创建一个 Dog 对象。但是因为 getAnimal 的返回类型为 Animal，所以我们直接得到的只是一个 Animal 对象，这时可以使用

cast 操作符将其转化为 Dog 对象，参见代码 2-2-6。

代码 2-2-6

```
class Animal {
    protected String name = "Animal";
    Animal() {
        log("create " + name);
    }
    String getName() {
        return name;
    }
}
class Dog extends Animal {
    Dog() {
        name = getClass().getSimpleName();
        log("create " + name);
    }
}

Animal getAnimal() {
    return new Dog();
}

private Observable<Dog> castObserver() {
    return Observable.just(getAnimal())
            .cast(Dog.class);
}

castObserver().subscribe(new Action1<Dog>() {
    @Override
    public void call(Dog dog) {
        log("cast:" + dog.getName());
    }
});
```

运行后得到如下结果。可以看到输出了 Dog 的类名，说明 cast 操作符将 Animal 类型的对象强制转化成为 Dog 类型的对象。另外我们还可以验证一个小知识点：有继承的情况下创建对象会首先调用父类的构造方法，然后调用子类的构造方法。

```
create Animal
create Dog
cast:Dog
```

2.2.6　scan

scan 操作符对一个序列的数据应用同一个函数进行计算，并将这个函数的结果发送出去，作为下一个数据应用这个函数时的第一个参数使用。也就是说任何一个要发送出来的数据都是以上次发送出来的数据和这次源 Observable 发送的数据作为输入，并应用同一个函数计算后的结果。所以 scan 操作符非常类似于递归操作符，其示意图如图 2-2-7 所示。由于 scan 所使用的这个函数需要输入两个数据作为参数，而第一个发送出来的数据只有一个，所以不会对第一个数据做任何计算，会直接将它发送出来，并作为一个参数和第二个数据一同传入函数中进行计算。

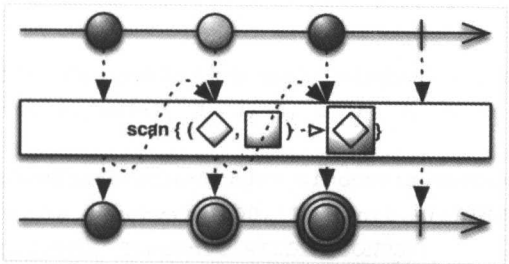


图 2-2-7

代码 2-2-7 通过一个存放 10 个 2 的 list 创建一个 Observable 对象，并使用 scan 操作符对其进行转化，转化的函数会让两个参数相乘。

代码 2-2-7

```
ArrayList<Integer> list = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    list.add(2);
}

private Observable<Integer> scanObserver() {
```

```
    return Observable.from(list).scan(new Func2<Integer, Integer, Integer>() {
        @Override
        public Integer call(Integer x, Integer y) {
            return x * y;
        }
    });
}

scanObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("scan:" + i);
    }
});
```

订阅后的输出结果如下，通过 scan 操作符我们输出了 2 的 *n* 次方。

```
scan:2
scan:4
scan:8
scan:16
scan:32
scan:64
scan:128
scan:256
scan:512
scan:1024
```

2.2.7 window

window 操作符类似于 2.2.1 节里的 buffer，不同之处在于 buffer 是将收集的数据打包作为一个整体发送出去，如发送一个 List 等，而 window 发送的是一些小的 Observable 对象，每个小的 Observable 对象包含 window 操作符规定的窗口里所收集到的数据，然后由这些小的 Observable 对象将其内部包含的数据一个个发送出来。如同 buffer 一样，window 不仅可以通过数目来分组，还可以通过时间等规则来分组，其示意图如图 2-2-8 所示。

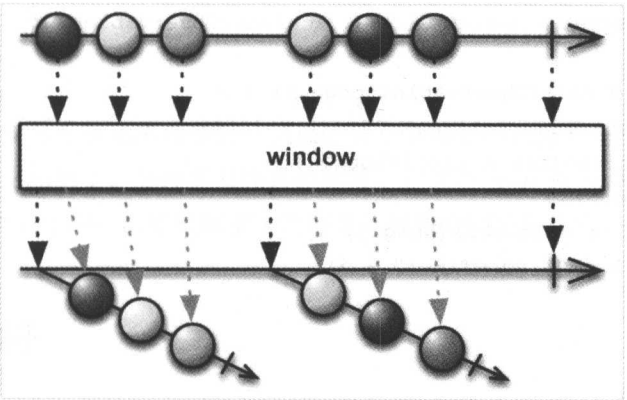


图 2-2-8

下面我们创建两个 Observable 对象，分别使用 window 的数目和时间规则来进行分组并进行订阅。因为 window 操作符发送出来的数据是一些小的 Observable，所以我们还会对这些小的 Observable 进行订阅，如代码 2-2-8 所示。

代码 2-2-8

```
private Observable<Observable<Integer>> windowCountObserver() {
    return Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9).window(3);
}
private Observable<Observable<Long>> windowTimeObserver() {
    return Observable.interval(1000, TimeUnit.MILLISECONDS)
            .window(3000, TimeUnit.MILLISECONDS);
}
windowCountObserver()
        .subscribe(new Action1<Observable<Integer>>() {
            @Override
            public void call(Observable<Integer> i) {
                log(i.getClass().getName());
                i.subscribe(new Action1<Integer>() {
                    @Override
                    public void call(Integer j) {
                        log("window:" + j);
                    }
                });
            }
        });
```

```
windowTimeObserver().subscribe(new Action1<Observable<Long>>() {
    @Override
    public void call(Observable<Long> i) {
        log(System.currentTimeMillis() / 1000);
        i.subscribe(new Action1<Long>() {
            @Override
            public void call(Long j) {
                log("windowTime:" + j);
            }
        });
    }
});
```

输出的结果如下所示。我们可以看到前一个 Observable 里面的 9 个数据被 window 操作符分成了 3 个小的 Observable，每个小的 Observable 各自会发送出 3 个数据。而后一个 Observable 会每隔 3 秒发送出一个小的 Observable，每个小的 Observable 会发送其收集的 2 个或 3 个数据。

```
rx.subjects.UnicastSubject
window:1
window:2
window:3
rx.subjects.UnicastSubject
window:4
window:5
window:6
rx.subjects.UnicastSubject
window:7
window:8
window:9
1500088005
windowTime:0
windowTime:1
1500088008
windowTime:2
windowTime:3
windowTime:4
1500088011
windowTime:5
```

```
windowTime:6
windowTime:7
```

转化操作符可以说是平常运用最多的一类操作符，掌握好转化操作符的使用技巧基本就可以处理好大多数的应用场景了。另外需要注意的就是，记清楚 map 和 flatMap，以及 buffer 和 window 等操作符之间的区别，以便根据实际的需要选择最合适的操作符。

2.3　过滤操作符

在 2.2 节里，我们了解了转化操作符，它们能将数据转化为我们想要的格式。但是如果数据集合里面有一些我们想要过滤掉的数据该怎么办？这时候就需要使用过滤操作符了，过滤操作符的作用类似于 SQL 里的 where 语句，能够让 Observable 只发送出满足条件的数据。

2.3.1　debounce

debounce 操作符是用来做限流的，可以将其理解为一个阀门：当半开阀门的时候，水会以较慢的速度流出来。不同之处就是阀门里流出的水一般不会被直接丢掉，而 debounce 过滤掉的数据会被丢弃掉。在 RxJava 中，这个操作符又被分为了 throttleWithTimeout 和 debounce 两个操作符，如图 2-3-1 和图 2-3-2 所示。

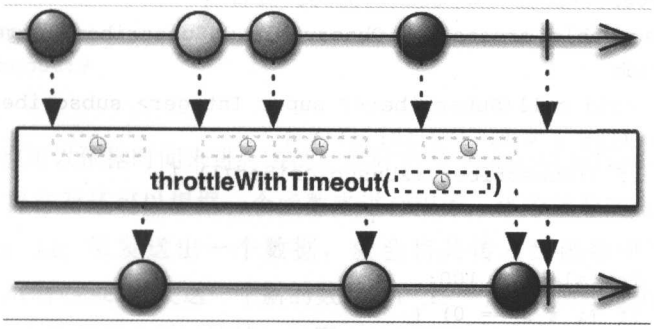


图 2-3-1

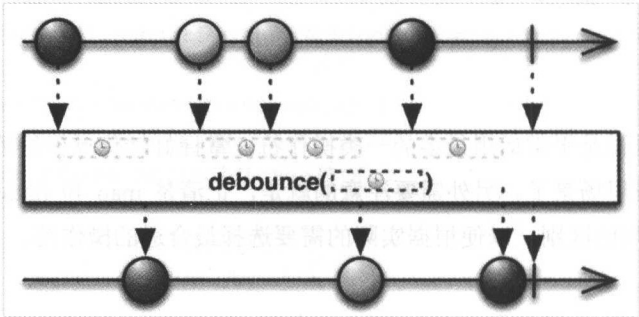


图 2-3-2

先来看一下 throttleWithTimeout 吧，这个操作符通过时间来限流，每次接收到源 Observable 发送出来的一个数据后就会进行计时，如果在设定好的时间结束前又接收到源 Observable 发送的新数据，那么上一个数据就会被丢弃，同时重新开始计时。如果每次都是在计时结束前接收到新数据，那么这个限流就会走向极端：只发送最后一个数据。

下面我们来创建一个 Observable，它会每隔 100 毫秒发送一个数据，当要发送的数据是 3 的倍数时，下一个数据就延迟到 300 毫秒后再发送。我们使用 throttleWithTimeout 来过滤一下这个 Observable，设定过滤时间为 200 毫秒，也就是说发送间隔小于 200 毫秒的数据会被过滤掉，如代码 2-3-1 所示。

代码 2-3-1

```
private Observable<Integer> createObserver() {
    return Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            for (int i = 0; i < 10; i++) {
                if (!subscriber.isUnsubscribed()) {
                    subscriber.onNext(i);
                }
                int sleep = 100;
                if (i % 3 == 0) {
                    sleep = 300;
                }
                try {
                    Thread.sleep(sleep);
                } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }
        subscriber.onCompleted();
    }
}).subscribeOn(Schedulers.computation());
}
private Observable<Integer> throttleWithTimeoutObserver() {
    return createObserver().throttleWithTimeout(200, TimeUnit.MILLISECONDS);
}

throttleWithTimeoutObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("throttleWithTimeout:" + i);
    }
});
```

订阅后来查看一下输出的结果，可以看到，对于不是 3 的倍数的数据，在其被发送出来 100 毫秒后，Observable 又会发送新的数据，因为 100 毫秒小于过滤时间间隔，所以这些数据会被过滤掉，最终只有 3 的倍数被发送了出来：

```
throttleWithTimeout:0
throttleWithTimeout:3
throttleWithTimeout:6
throttleWithTimeout:9
```

debounce 操作符也可以依据时间来进行过滤，这时它跟 throttleWithTimeout 使用起来是一样的，但是 deounce 操作符还可以根据一个函数来进行限流。这个函数的返回值是一个临时 Observable，源 Observable 每发送出一个数据，就会将其传入到函数中产生一个临时的 Observable。如果源 Observable 在发送一个新的数据时，上一个数据所生成的临时 Observable 还没有结束，那么上一个数据就会被过滤掉，如图 2-3-3 所示。

在代码 2-3-2 中我们使用 interval 操作符生成一个 Observable，它会每隔 1000 毫秒发送出一个数据，然后我们在 debounce 的函数里，使用 timer 操作符来创建临时 Observable。对于偶数，timer 创建的临时操作符会马上结束；而对于奇数，timer 将会在 1500 毫秒之后

才结束。因为1500毫秒大于1000毫秒，所以所有的奇数都会被过滤掉，最终只能输出偶数。

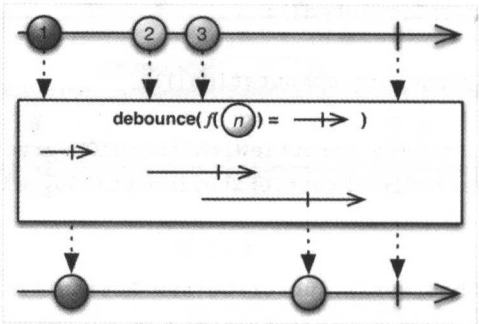


图 2-3-3

代码 2-3-2

```
private Observable<Long> debounceObserver() {
    return Observable.interval(1000, TimeUnit.MILLISECONDS)
            .debounce(new Func1<Long, Observable<Long>>() {
                @Override
                public Observable<Long> call(Long aLong) {
                    return Observable.timer(aLong % 2 * 1500,
                            TimeUnit.MILLISECONDS);
                }
            });
}
```

运行程序后输出的结果如下，可以看到经过 debounce 过滤后所有的奇数都被过滤掉了，只有偶数被发送了出来：

```
debounce:0
debounce:2
debounce:4
debounce:6
debounce:8
...
...
...
```

2.3.2　distinct

distinct 操作符的用处就是去重，非常好理解。如图 2-3-4 所示，所有重复的数据都会被过滤掉，其作用范围是全局的，也就是说所有发送出来的数据肯定是没有重复的。distinct 还有一个特殊的实现操作符：distinctUntilChanged，这个操作符被用来过滤掉连续的重复数据，其作用范围是局部的，也就是说发送出来的数据可能会有重复，但是这些重复的数据肯定是不连续的，其示意图如图 2-3-5 所示。

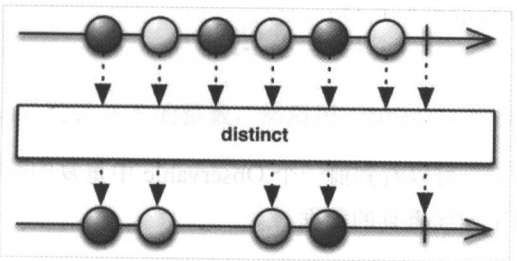


图 2-3-4

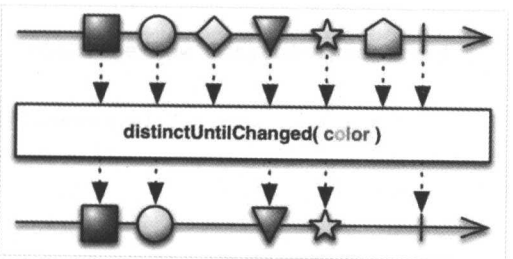


图 2-3-5

下面创建两个有重复数据的 Observable，并使用 distinct 和 distinctUntilChanged 操作符分别对其进行过滤，如代码 2-3-3 所示。

代码 2-3-3

```
private Observable<Integer> distinctObserver() {
    return Observable.just(1, 2, 3, 4, 5, 4, 3, 2, 1).distinct();
}
private Observable<Integer> distinctUntilChangedObserver() {
    return Observable.just(1, 2, 3, 3, 3, 1, 2, 3, 3).distinctUntilChanged();
}
```

```
distinctObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("distinct:" + i);
    }
});
distinctUntilChangedObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("UntilChanged:" + i);
    }
});
```

订阅后的输出结果如下，可以看到前一个 Observable 中重复的数据都被过滤掉了，而后一个 Observable 中仅过滤掉了连续重复的数据。

```
distinct:1
distinct:2
distinct:3
distinct:4
distinct:5
UntilChanged:1
UntilChanged:2
UntilChanged:3
UntilChanged:1
UntilChanged:2
UntilChanged:3
```

2.3.3 elementAt

elementAt 只会过滤出源 Observable 发送出来的顺序为 N 的数据。同数组的下标一样，这里的 N 也是从 0 开始的，也就说如果 N 为 0，将只会过滤出源 Observable 发送的第一个数据，依此类推，其示意图如图 2-3-6 所示。

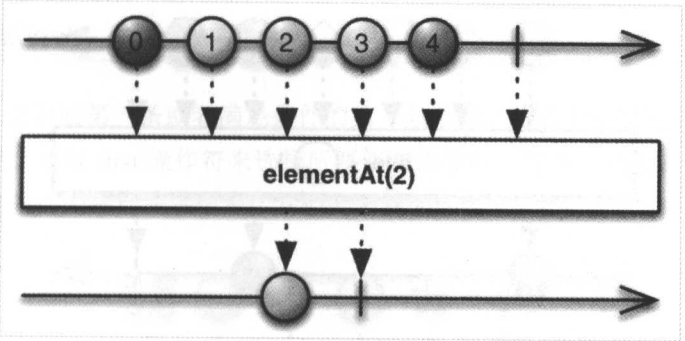


图 2-3-6

下面创建一个 Observable 发送 0~5 的整数，然后用 elementAt 操作符将顺序为 2 的数据过滤出来，如代码 2-3-4 所示。

代码 2-3-4

```
private Observable<Integer> elementAtObserver() {
    return Observable.just(0, 1, 2, 3, 4, 5).elementAt(2);
}
elementAtObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("elementAt:" + i);
    }
});
```

订阅后查看其输出结果如下。

```
elementAt:2
```

2.3.4　filter

filter 操作符根据一个函数来进行过滤操作。源 Observable 发送的数据作为参数传递到函数里，如果函数的返回值为 true，则这个数据将继续被发送出去，否则数据将会被过滤掉，其示意图如图 2-3-7 所示。

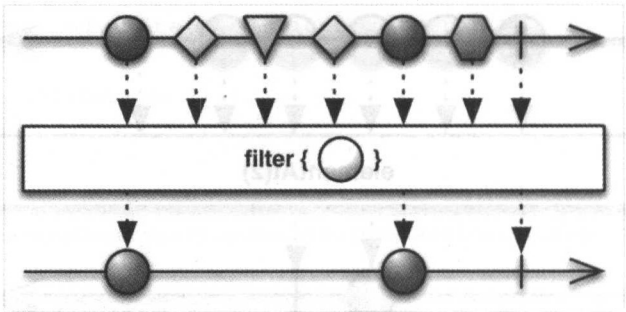


图 2-3-7

下面创建一个 Observable 发送 0~5 的整数，并使用 filter 操作符进行过滤。过滤的规则是只留下小于 3 的数，如代码 2-3-5 所示。

代码 2-3-5

```
private Observable<Integer> filterObserver() {
    return Observable.just(0,1,2,3,4,5).filter(new Func1<Integer, Boolean>() {
        @Override
        public Boolean call(Integer integer) {
            return integer < 3;
        }
    });
}
filterObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("Filter:" + i);
    }
});
```

订阅后的结果如下所示。只输出了 0、1、2，其他的数据都被过滤掉了。

```
Filter:0
Filter:1
Filter:2
```

2.3.5　first 和 last

first 操作符只会返回第一条或者满足条件的第一条数据。在第 3 章的使用 RxJava 实现三级缓存的例子里，就会使用 first 操作符来选择所要使用的缓存。与 first 相反，last 操作符只返回最后一条或者满足条件的最后一条数据，它们的示意图如图 2-3-8 和图 2-3-9 所示。

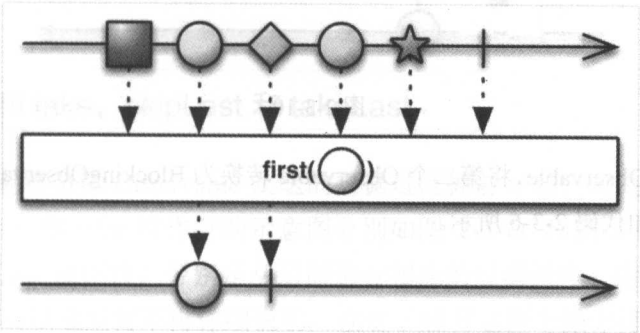


图 2-3-8

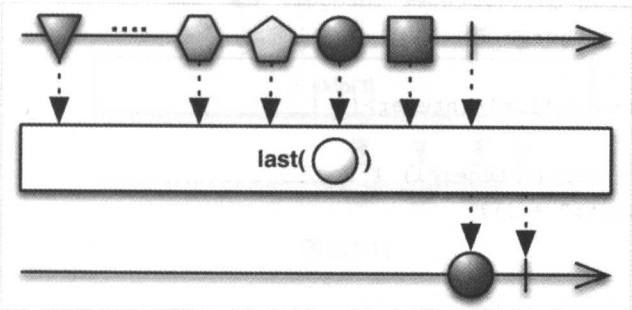


图 2-3-9

另外还有一个特殊的 Observable：BlockingObservable，可以使用 Observable.toBlocking 或者 BlockingObservable.from 方法来将一个 Observable 对象转化为 BlockingObservable 对象。BlockingObservable 不会对 Observable 做任何处理，只会阻塞住。只有当满足条件的数据发送出来的时候才会发送出一个 BlockingObservable 对象。BlockingObservable 也有可以配合使用的 first 操作符，不过其 first 操作符返回的不是 Observable，而是对应的数据，如示意图 2-3-10 所示。

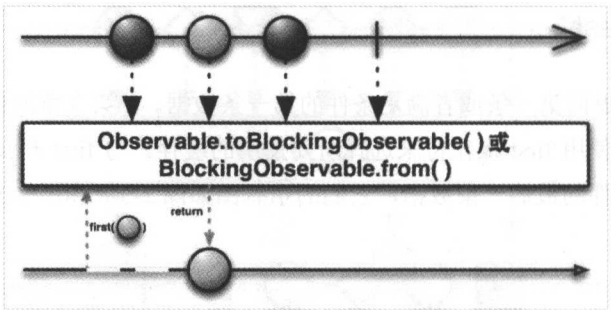


图 2-3-10

下面创建两个 Observable，将第二个 Observable 转换为 BlockingObservable，然后分别用 first 操作符进行处理，如代码 2-3-6 所示。

代码 2-3-6

```
Observable.just(0, 1, 2, 3, 4, 5).first(new Func1<Integer, Boolean>() {
    @Override
    public Boolean call(Integer integer) {
        return integer > 3;
    }
}).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("First:" + i);
    }
});

int result = Observable.just(0, 1, 2, 3, 4, 5)
        .toBlocking()
        .first(new Func1<Integer, Boolean>() {
            @Override
            public Boolean call(Integer i) {
                return i > 1;
            }
        });
log("blocking:" + result);
```

输出的结果如下。两个 Observable 经过 first 操作符过滤后分别输出了第一个大于 3 的数和

第一个大于 1 的数。

```
first:4
blocking:2
```

last 操作符的使用方法同 first 基本一样，所以这里不再赘述。

2.3.6　skip 和 take，skipLast 和 takeLast

skip 操作符能够将源 Observable 发送的数据过滤掉前 *n* 项，与之相反，take 操作符则只取前 *n* 项，skip 和 take 操作符的示意图分别如图 2-3-11 和图 2-3-12 所示。另外还有 skipLast 和 takeLast 操作符，分别是从后面进行相应的过滤操作，由于它们的功能和 skip 和 take 很类似，所以这里将不再详细说明，读者了解有这两个操作符即可。

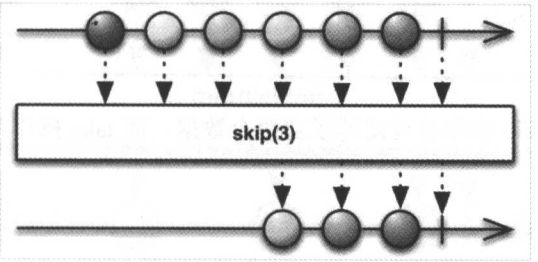


图 2-3-11

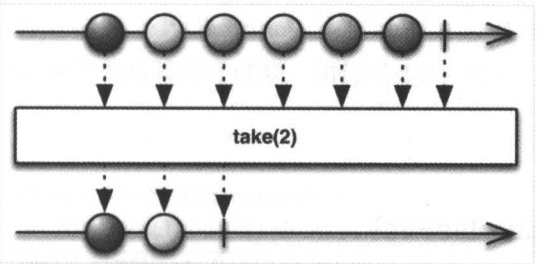


图 2-3-12

下面创建两个 Observable 并分别使用 skip 和 take 操作符对其进行过滤操作，之后进行订阅查看输出结果，如代码 2-3-7 所示。

代码 2-3-7

```
private Observable<Integer> skipObserver() {
    return Observable.just(0, 1, 2, 3, 4, 5).skip(2);
}
private Observable<Integer> takeObserver() {
    return Observable.just(0, 1, 2, 3, 4, 5).take(2);
}
skipObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("skip:" + i);
    }
});
takeObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("take:" + i);
    }
});
```

输出的结果如下，skip 操作符过滤掉了前两个数据，而 take 操作符则只取了前两个数，将后面的数都过滤掉了。

```
skip:2
skip:3
skip:4
skip:5
take:0
take:1
```

2.3.7 sample 和 throttleFirst

sample 操作符会指定一段时长，在每段时长结束的时候发送源 Observable 发送的最新数据，其他的都会被过滤掉。也就是说在 sample 指定的这段时间内无论源 Observable 发送了多少数据，只有最后一个数据才能通过 sample 的过滤。sample 操作符的示意图如图 2-3-13 所示。

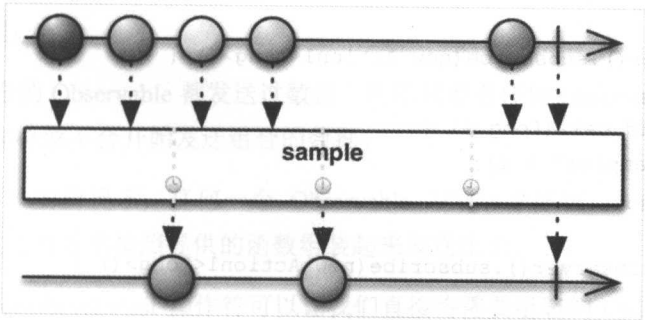


图 2-3-13

sample 是取规定时间段内最后一个数据，而 throttleFirst 操作符取的则是规定时间段内的第一个数据，其他的会被过滤掉。在 Android 开发中，经常会有过滤掉过多的点击事件的需求，使用 throttleFirst 就可以很容易地将过多的点击事件过滤掉，其示意图如图 2-3-14 所示。

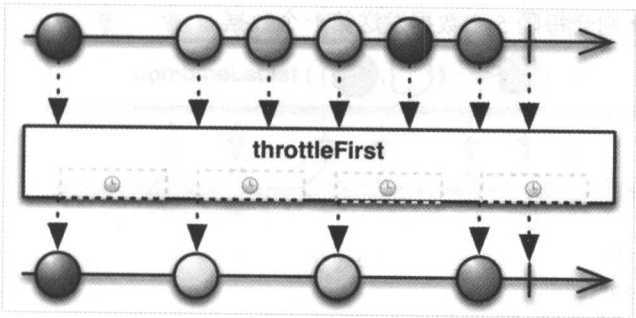


图 2-3-14

接下来使用 interval 操作符创建一个 Observable，它会每隔 200 毫秒发送一个数据，然后分别使用 sample 和 throttleFirst 操作符对其进行过滤，如代码 2-3-8 所示。

代码 2-3-8

```
private Observable<Long> sampleObserver() {
    return Observable.interval(200, TimeUnit.MILLISECONDS)
            .sample(1000, TimeUnit.MILLISECONDS);
}
private Observable<Long> throttleFirstObserver() {
    return Observable.interval(200, TimeUnit.MILLISECONDS)
            .throttleFirst(1000, TimeUnit.MILLISECONDS);
```

```
}
sampleObserver().subscribe(new Action1<Long>() {
    @Override
    public void call(Long i) {
        log("sample:" + i);
    }
});
throttleFirstObserver().subscribe(new Action1<Long>() {
    @Override
    public void call(Long i) {
        log("throttleFirst:" + i);
    }
});
```

订阅后，sample 的输出结果如下，可以看到 sample 操作符会每隔 5 个数字发送出最后一个数据，而 throttleFirst 则会每隔 5 个数据发送第一个数据。

```
sample:3
sample:8
sample:13
sample:18
throttleFirst:0
throttleFirst:5
throttleFirst:10
throttleFirst:15
```

2.4 组合操作符

组合操作符会将多个 Observable 发送的数据按照一定的规则组合起来，这在汇总各种结果的时候就显得非常有用了。

2.4.1 combineLatest

combineLatest 操作符可以将 2~9 个 Observable 发送的数据组合起来，然后再发送出来。不

过还需要两个前提：

1. 所有要组合的 Observable 都发送过数据。所以只要有任何 Observable 还未发送过数据，combineLatest 操作符就不会开始发送组合的数据。

2. 满足条件 1 的前提下，任何一个 Observable 发送一个数据，combineLatest 就将所有 Observable 最新发送的数据按照提供的函数组装起来发送出去。

RxJava 实现 combineLatest 操作符可以让我们直接将需要组装的 Observable 对象作为参数传值，也可以将所有的 Observable 装在一个 List 里面传进去，其示意图如图 2-4-1 所示。

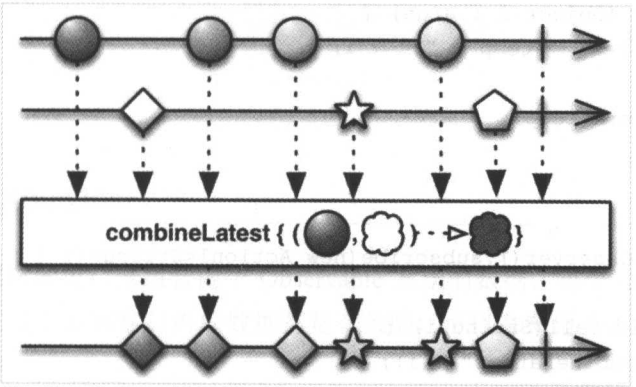


图 2-4-1

下面我们使用 interval 分别创建两个 Observable 对象，间隔分别是 500 毫秒和 1000 毫秒，然后分别使用直接传值和 List 传值的方式将其组装起来，如代码 2-4-1 所示，对其进行订阅查看输出结果。

代码 2-4-1

```
private Observable<Long> createObserver(int index) {
    return Observable.interval(500 * index, TimeUnit.MILLISECONDS);
}
private Observable<String> combineLatestObserver() {
    return Observable.combineLatest(createObserver(1), createObserver(2),
            new Func2<Long, Long, String>() {
                @Override
                public String call(Long num1, Long num2) {
                    return ("left:" + num1 + " right:" + num2);
```

```
                }
            });
    }
    private Observable<String> combineListObserver() {
        for (int i = 1; i < 3; i++) {
            list.add(createObserver(i));
        }
        return Observable.combineLatest(list, new FuncN<String>() {
            @Override
            public String call(Object... args) {
                String temp = "";
                for (Object i : args) {
                    temp = temp + ":" + i;
                }
                return temp;
            }
        });
    }
    combineLatestObserver().subscribe(new Action1<String>() {
        @Override
        public void call(String i) {
            log("CombineLatest" + i);
        }
    });
    combineListObserver().subscribe(new Action1<String>() {
        @Override
        public void call(String i) {
            log("combineList:" + i);
        }
    });
```

查看一下输出的结果，无论是将 Observable 对象直接传值还是放到 List 里面传值，由于两个 Observable 发送数据的间隔时间不同，导致前一个 Observable 发送的数据较快，所以在进行组合操作的时候就会出现两三个数据都和同一个数据组装的现象。这是由 combineLatest 操作符的特性决定的：只组装每个 Observable 最新的数据，而不管这个数据是否已经组装过了。在使用 combineLatest 操作符的时候一定要注意这一点。

```
CombineLatestleft:1 right:0
CombineLatestleft:2 right:0
CombineLatestleft:3 right:0
CombineLatestleft:3 right:1
CombineLatestleft:4 right:1
CombineLatestleft:5 right:1
CombineLatestleft:5 right:2
combineList::1:0
combineList::2:0
combineList::3:0
combineList::3:1
combineList::4:1
combineList::5:1
combineList::5:2
```

2.4.2　join 和 groupJoin

join 操作符根据时间窗口来组合两个 Observable 发送的数据，每个 Observable 都有一个自己的时间窗口，在这个时间窗口内的数据都是有效的，可以拿来组合。其示意图如图 2-4-2 所示。

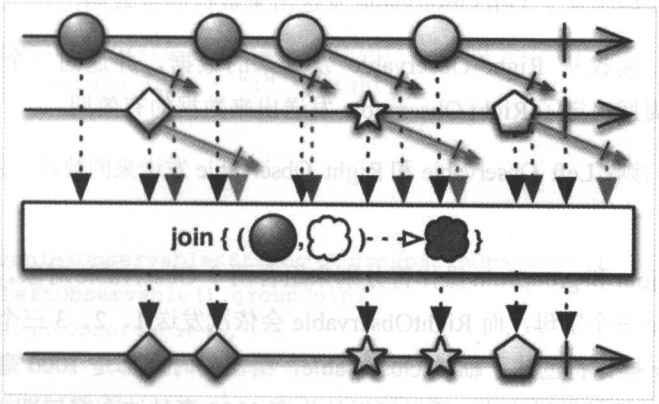


图 2-4-2

RxJava 还实现了 groupJoin，它和 join 基本相同，只是通过 groupJoin 操作符组合后，发送出来的是一个个小的 Observable，每个 Observable 里面包含了一轮组合数据。这样讲可能不是

很好理解，可以结合后面的代码 2-4-2 来帮助理解，groupJoin 的示意图如图 2-4-3 所示。

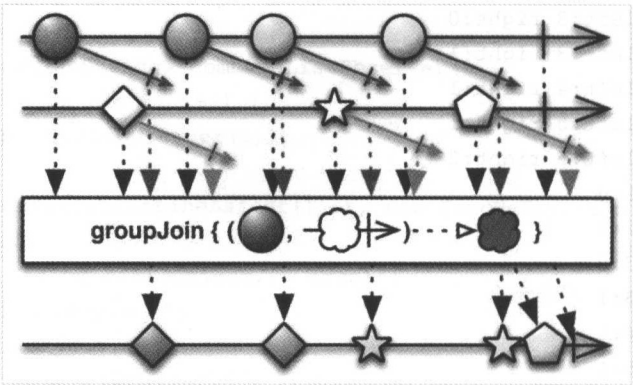


图 2-4-3

join 操作符需要对两个 Observable 进行操作，为了区别，我们用左（Left）和右（Right）来区分要组合的这两个 Observable，即 Left Observable 和 Right Observable。使用 join 操作符需要 4 个参数，它们分别是：

- 要被组合到 Left Observable 的 Right Observable。
- 一个函数，接收从 Left Observable 发送来的数据，并返回一个 Observable，这个 Observable 的生命周期决定了 Left Observable 发送出来数据的有效期。
- 一个函数，接收从 Right Observable 发送来的数据，并返回一个 Observable，这个 Observable 的生命周期决定了 Right Observable 发送出来数据的有效期。
- 一个函数，接收 Left Observable 和 Right Observable 发送来的数据，并返回最终组合完的数据。

下面我们使用 join 和 groupJoin 操作符分别来组合两个 Observable 对象。其中 LeftObservable 会依次发送 a、b、c 三个字母，而 RightObservable 会依次发送 1、2、3 三个数据。在组合的时候，我们使用 timer 操作符创建了临时 Observable，给出的时间都是 1000 毫秒。因为 just 操作符会很快把所有的数据都发送出来，所以可以认为在 1000 毫秒这个窗口期内，所有的数据都是有效的。组合完毕后订阅查看输出，如代码 2-4-2 所示。

代码 2-4-2

```
private Observable<String> getLeftObservable() {
    return Observable.just("a", "b", "c");
}
private Observable<Long> getRightObservable() {
    return Observable.just(11, 21, 31);
}
private Observable<String> joinObserver() {
    return getLeftObservable().join(
            getRightObservable(),
            new Func1<String, Observable<Long>>() {
                @Override
                public Observable<Long> call(String s) {
                    return Observable.timer(1000, TimeUnit.MILLISECONDS);
                }
            },
            new Func1<Long, Observable<Long>>() {
                @Override
                public Observable<Long> call(Long s) {
                    return Observable.timer(1000, TimeUnit.MILLISECONDS);
                }
            },
            new Func2<String, Long, String>() {
                @Override
                public String call(String left, Long right) {
                    return left + ":" + right;
                }
            }
    );
}
private Observable<Observable<String>> groupJoinObserver() {
    return getLeftObservable().groupJoin(
            getRightObservable(),
            new Func1<String, Observable<Long>>() {
                @Override
                public Observable<Long> call(String s) {
                    return Observable.timer(1000, TimeUnit.MILLISECONDS);
                }
            },
```

```
                new Func1<Long, Observable<Long>>() {
                    @Override
                    public Observable<Long> call(Long s) {
                        return Observable.timer(1000, TimeUnit.MILLISECONDS);
                    }
                },
                new Func2<String, Observable<Long>, Observable<String>>() {
                    @Override
                    public Observable<String> call(String left, Observable<Long>
longObservable) {
                        return longObservable.map(new Func1<Long, String>() {
                            @Override
                            public String call(Long right) {
                                return left + ":" + right;
                            }
                        });
                    }
                }
        );
    }
    joinObserver().subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
            log("join:" + s);
        }
    });
    stringObservable.first().subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
            log("groupJoin:" + s);
        }
    });
```

输出的结果如下。可以看到，因为所有的数据都是在有效期窗口内的，所以使用 join 操作符进行组合时，所有的数据都会一一进行组合。对于 groupJoin，我们使用了 first 操作符，从而只取第一个发送出来的小的 Observable，对这个小的 Observable 进行订阅只输出了一组的数据。如果不使用 first 操作符进行处理的话，输出的结果将和使用 join 的结果一样。

```
join:b:1
join:a:1
join:c:1
join:b:2
join:a:2
join:c:2
join:b:3
join:a:3
join:c:3
groupJoin:b:1
groupJoin:a:1
groupJoin:c:1
```

2.4.3　merge 和 mergeDelayError

merge 操作符将多个 Observable 发送的数据整合起来发送，对外就如同是一个 Observable 发送的数据一样，其示意图如图 2-4-4 所示。但是其发送的数据有可能是有交错的，如果不想有交错，可以使用 concat 操作符。

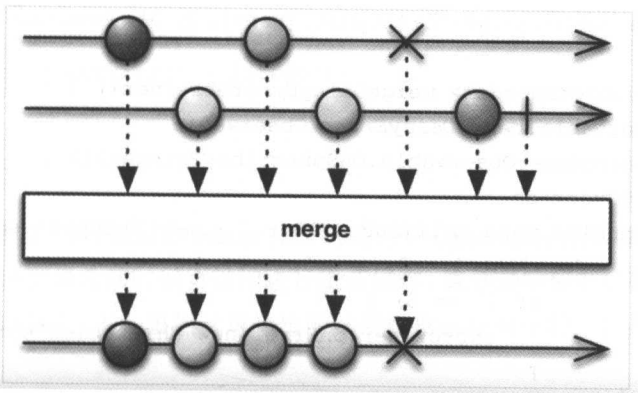


图 2-4-4

当某一个 Observable 发出 onError 的时候，merge 的过程会被停止并将错误分发给 Subscriber，如果不想让错误中止 merge 的过程，可以使用 meregeDelayError 操作符，它会在 merge 结束后再分发错误。meregeDelayError 的示意图如图 2-4-5 所示。

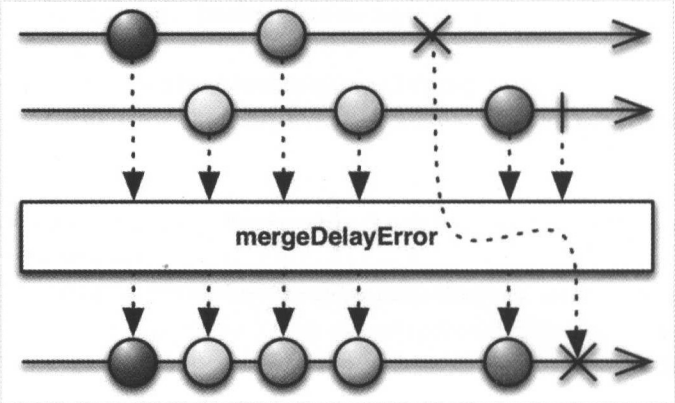


图 2-4-5

下面我们分别使用 merge 和 mergeDelayError 操作符来进行 merge 操作。对于使用 mergeDelayError 中的一个 Observable，当发送的数据为 3 时，它会抛出一个错误用来验证 mergeDelayError 的作用。

代码 2-4-3

```
private Observable<Integer> mergeObserver() {
    return Observable.merge(Observable.just(1, 2, 3), Observable.just(4, 5, 6));
}
private Observable<Integer> mergeDelayErrorObserver() {
    return Observable.mergeDelayError(Observable
            .create(new Observable.OnSubscribe<Integer>() {
                @Override
                public void call(Subscriber<? super Integer> subscriber) {
                    for (int i = 0; i < 5; i++) {
                        if (i == 3) {
                            subscriber.onError(new Throwable("error"));
                        }
                        subscriber.onNext(i);
                    }
                }
            }), Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            for (int i = 0; i < 5; i++) {
```

```
                subscriber.onNext(5 + i);
            }
            subscriber.onCompleted();
        }
    }));
}
mergeObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("Merge:" + i);
    }
});
mergeDelayErrorObserver().subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        log("onCompleted");
    }
    @Override
    public void onError(Throwable e) {
        log("mergeDelayError:" + e);
    }
    @Override
    public void onNext(Integer integer) {
        log("mergeDelayError:" + integer);
    }
});
```

订阅后的输出结果如下。可以看到 merge 操作符将两个 Observable 的数据依次发送了出来。而 mergeDelayError 在碰到异常的时候会继续进行组合操作，直到最后所有的数据都发送完毕后才抛出这个异常。再次提醒一下，虽然这里输出的结果数据都是有序的，但是 merge 不能保证数据的有序性。

```
Merge:1
Merge:2
Merge:3
Merge:4
Merge:5
Merge:6
```

```
mergeDelayError:0
mergeDelayError:1
mergeDelayError:2
mergeDelayError:3
mergeDelayError:4
mergeDelayError:5
mergeDelayError:6
mergeDelayError:7
mergeDelayError:8
mergeDelayError:9
mergeDelayError:java.lang.Throwable: error
```

2.4.4 startWith

startWith 操作符会在源 Observable 发送的数据前面插入一些数据。startWith 不仅可以插入一些数据，还可以将 Iterable 和 Observable 插入进去。如果插入的是 Observable，则这个 Observable 发送的数据会插入到源 Observable 发送的数据前面，其示意图如图 2-4-6 所示。

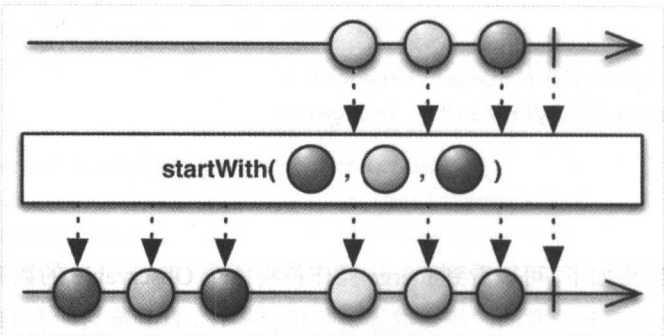


图 2-4-6

创建一个 Observable 并使用 startWith 操作符在其前面插入两个数据，如代码 2-4-4 所示。

代码 2-4-4

```
private Observable<Integer> startWithObserver() {
    return Observable.just(1, 2, 3).startWith(-1, 0);
}
startWithObserver().subscribe(new Action1<Integer>() {
```

```
    @Override
    public void call(Integer i) {
        log("StartWith:" + i);
    }]
});
```

订阅后的输出结果如下。可以看到 –1 和 0 被插入到了 1、2、3 的前面，并且被先发送了出来。

```
StartWith:-1
StartWith:0
StartWith:1
StartWith:2
StartWith:3
```

2.4.5　switch

在 RxJava 中，源 Observable 发送出来的数据可能是一个个小的 Observable，如果订阅者只对这些小的 Observable 所发送出来的数据感兴趣，就需要使用 switch 操作符。switch 操作符在 RxJava 上的实现为 switchOnNext，用来将源 Observable 发送出来的多个小 Observable 组合为一个 Observable，然后发送这些多个小 Observable 所发送的数据。需要注意的是，如果一个小的 Observable 正在发送数据时，源 Observable 又发送出一个新的小 Observable，则前一个 Observable 未发送的数据会被抛弃，直接发送新的小 Observable 所发送的数据。如图 2-4-7 所示，斜向箭头上有三个圆圈中，最右端的圆圈所代表的数据被丢弃了。

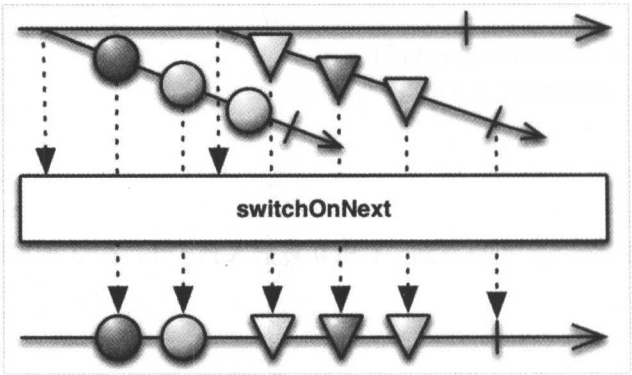


图 2-4-7

创建一个 Observable，使它每隔 3 秒钟就发送出一个小的 Observable 对象，而这些小的 Observable 会每隔 1 秒就依次发送 0、1、2、3、4 这几个数字。

代码 2-4-5

```
private Observable<String> createObserver(Long index) {
    return Observable.interval(1000, 1000, TimeUnit.MILLISECONDS).take(5)
            .map(new Func1<Long, String>() {
                @Override
                public String call(Long aLong) {
                    return index + "-" + aLong;
                }
            });
}
private Observable<String> switchObserver() {
    return Observable.switchOnNext(Observable
            .interval(3000, TimeUnit.MILLISECONDS)
            .take(3)
            .map(new Func1<Long, Observable<String>>() {
                @Override
                public Observable<String> call(Long aLong) {
                    return createObserver(aLong);
                }
            }));
}
switchObserver().subscribe(new Action1<String>() {
    @Override
```

```
    public void call(String s) {
        log("switch:" + s);
    }
});
```

订阅后的输出结果如下。前面的数字代表了小 Observable 的序号，后面的数字代表了小 Observable 发送出的数据。可以看到前面序号是 0 和 1 的小 Observable 都只发送出了两个数据，这是因为它们发送了两个数据后，新的小 Observable 就被发送出来了，所以序号是 0 和 1 的小 Observable 未发送的数据被丢弃了。而对于序号为 2 的小 Observable，因为其后面没有新的小 Observable 被发送出来，所以它可以将所有的数据都发送出来。

```
switch:0-0
switch:0-1
switch:1-0
switch:1-1
switch:2-0
switch:2-1
switch:2-2
switch:2-3
switch:2-4
```

2.4.6　zip 和 zipWith

zip 操作符将多个 Observable 发送的数据按顺序组合起来。与 join 操作符不同的是，join 中的每个数据可以组合多次，zip 操作符里的每个数据只能组合一次，而且都是有序的。最终组合的数据的数量由发送数据最少的 Observable 来决定。RxJava 实现了 zip 和 zipWith 两个操作符，示意图如图 2-4-8 和图 2-4-9 所示。

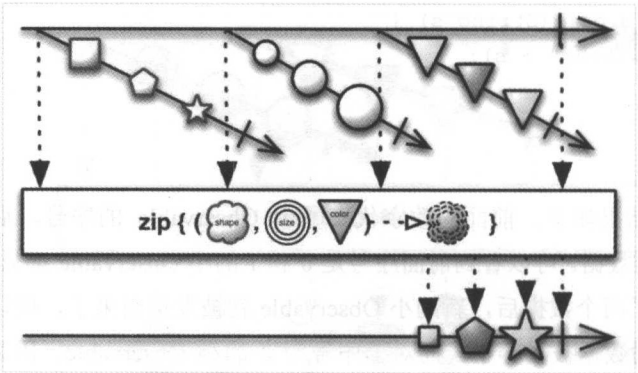


图 2-4-8

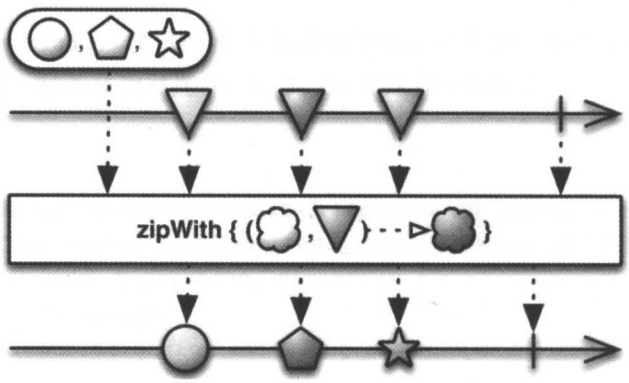


图 2-4-9

首先创建一个 createObserver 方法，根据传入的 index 的数值返回一个会发送 index 个数据的 Observable 对象，然后使用 zipWith 来组合一个发送两个数据和一个发送三个数据的 Observable；使用 zip 操作符将分别发送二个、三个和四个数据的三个 Observable 对象组合起来，如代码 2-4-6 所示，分别进行订阅，查看输出结果。

代码 2-4-6

```
private Observable<String> zipWithObserver() {
    return createObserver(2).zipWith(createObserver(3),
            new Func2<String, String, String>() {
                @Override
                public String call(String s, String s2) {
                    return s + "-" + s2;
```

```
            }
        });
}
private Observable<String> zipWithIterableObserver() {
    return Observable.zip(createObserver(2),
createObserver(3), createObserver(4),
        new Func3<String, String, String, String>() {
            @Override
            public String call(String s, String s2, String s3) {
                return s + "-" + s2 + "-" + s3;
            }
        });
}
private Observable<String> createObserver(int index) {
    return Observable.interval(100, TimeUnit.MILLISECONDS).take(index)
        .map(new Func1<Long, String>() {
            @Override
            public String call(Long aLong) {
                return index + ":" + aLong;
            }
        });
}
zipWithObserver().subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
        log("zipWith:" + s);
    }
});
zipWithIterableObserver().subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
        log("zip:" + s);
    }
});
```

输出的结果如下。无论是 zipWith 还是 zip 操作符，最终都输出了两个组合后的数据，这是因为它们组合的 Observable 中包含一个只发送两个数据的 Observable，而每个数据最多只能组合一次，所以即使其他的 Observable 对象会发送多于两个的数据，也不会被组合进来了，可以用木桶理论来理解 zip。

```
zipWith:2:0-3:0
zipWith:2:1-3:1
zip:2:0-3:0-4:0
zip:2:1-3:1-4:1
```

2.5 错误处理操作符

RxJava 对错误的处理很方便，当有错误出现的时候就会调用 Subscriber 的 onError 方法将错误分发出去，由 Subscriber 自己来处理错误。这种处理错误的方式虽然方便，但也有缺点，就是每个 Subscriber 都要去定义如何处理错误，如果有 100 个 Subscriber 就要定义 100 遍。如何来统一地处理这些错误呢？在这一节中我们就来学习如何用错误处理操作符来集中统一地处理错误。

2.5.1 onErrorReturn

onErroReturn 操作符可以在发生错误时，让 Observable 发送一个预先定义好的数据并停止继续发送数据，正常结束整个过程，示意图如图 2-5-1 所示。

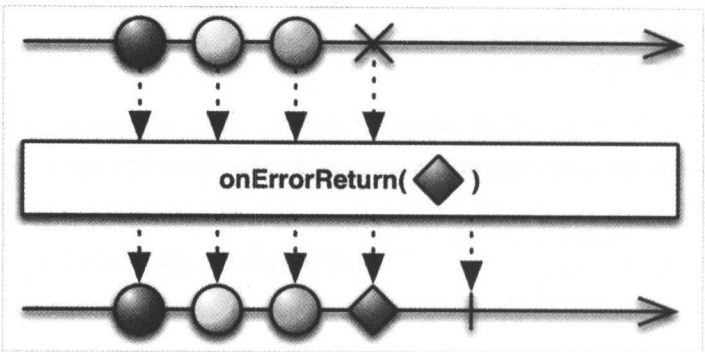


图 2-5-1

创建一个 Observable，发送 1~6 的整数，但是当数据为 3 时主动抛出一个异常。然后使用 onErrorReturn 来捕捉错误事件，当有错误事件时返回字符串"onErrorReturn"给订阅者，如代码 2-5-1 所示。

代码 2-5-1

```
private Observable<String> createObserver() {
    return Observable.create(new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            for (int i = 1; i <= 6; i++) {
                if (i < 3) {
                    subscriber.onNext("onNext:" + i);
                } else {
                    subscriber.onError(new Throwable("Throw error"));
                }
            }
        }
    });
}
private Observable<String> onErrorReturnObserver() {
    return createObserver().onErrorReturn(new Func1<Throwable, String>() {
        @Override
        public String call(Throwable throwable) {
            return "onErrorReturn";
        }
    });
}
onErrorReturnObserver().subscribe(new Subscriber<String>() {
    @Override
    public void onCompleted() {
        log("onErrorReturn-onCompleted");
    }
    @Override
    public void onError(Throwable e) {
        log("onErrorReturn-onError:" + e.getMessage());
    }
    @Override
    public void onNext(String s) {
        log("onErrorReturn-onNext:" + s);
    }
});
```

订阅后的输出结果如下，在发送出两个数据之后，onErrorReturn 捕捉到了抛出的异常并返回了指定的字符串给订阅者，然后整个 Observable 发送数据的过程结束。

```
onErrorReturn-onNext:onNext:1
onErrorReturn-onNext:onNext:2
onErrorReturn-onNext:onErrorReturn
onErrorReturn-onCompleted
```

2.5.2 onErrorResumeNext

onErrorReturn 操作符在有错误事件的时候会让 Observable 停止发送数据；而 onErrorResumeNext 在有错误发生的时候，会创建另外一个 Observable 来代替当前的 Observable 并继续发送数据，就好像错误并没有发生一样，其示意图如图 2-5-2 所示。

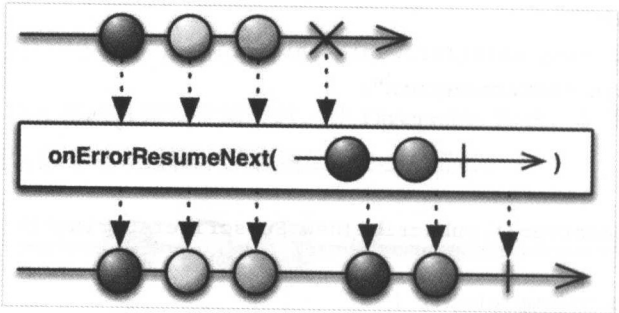


图 2-5-2

继续使用 2.4 节里的 createObserver 方法创建一个会产生错误事件的 Observable，然后使用 onErrorResumeNext 来捕捉错误事件，当有错误发生时创建一个新的 Observable 来继续发送数据，如代码 2-5-2 所示。

代码 2-5-2

```
private Observable<String> onErrorResumeNextObserver() {
    return createObserver().onErrorResumeNext(Observable.just("7", "8", "9"));
}
onErrorResumeNextObserver().subscribe(new Subscriber<String>() {
    @Override
```

```
        public void onCompleted() {
            log("onErrorResume-onCompleted");
        }
        @Override
        public void onError(Throwable e) {
            log("onErrorResume-onError:" + e.getMessage());
        }
        @Override
        public void onNext(String s) {
            log("onErrorResume-onNext:" + s);
        }
    });
```

订阅的输出结果如下，在正常发送两个数据后产生错误，新创建的 Observable 继续发送 7、8、9 三个数字，然后整个数据发送过程结束。

```
onErrorResume-onNext:onNext:1
onErrorResume-onNext:onNext:2
onErrorResume-onNext:7
onErrorResume-onNext:8
onErrorResume-onNext:9
onErrorResume-onCompleted
```

2.5.3　onExceptionResumeNext

onExceptionResumeNext 操作符类似于 onErrorResumeNext，不同之处在于其会对 onError 抛出的数据类型做判断——如果是 Exception，就会使用另外一个 Observable 代替原 Observable 继续发送数据，否则会将错误分发给 Subscriber，其示意图如图 2-5-3 所示。

创建一个方法，根据参数来决定错误事件发生的时候是抛出 Exception 还是 Throwable，然后创建一个方法，返回经过 onExceptionResumeNext 处理过的 Observable，如代码 2-5-3 所示。分别订阅查看抛出的错误类型为 Exception 时会输出什么结果，不为 Exception 时又会有什么不同。

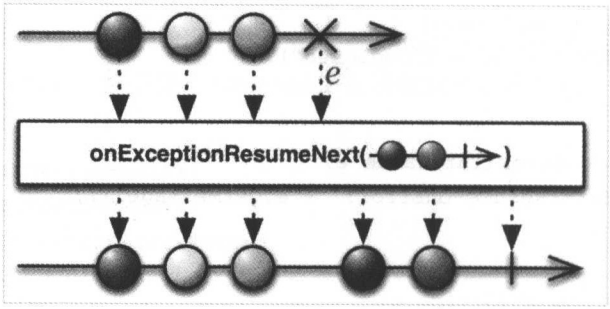


图 2-5-3

代码 2-5-3

```
    private Observable<String> createObserver(Boolean createExcetion) {
    return Observable.create(new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            for (int i = 1; i <= 6; i++) {
                if (i < 3) {
                    subscriber.onNext("onNext:" + i);
                } else if (createExcetion) {
                    subscriber.onError(new Exception("Exception"));
                } else {
                    subscriber.onError(new Throwable("Throw error"));

                }
            }
        }
    });
}
    private Observable<String> onExceptionResumeObserver(boolean isException) {
        return createObserver(isException)
                .onExceptionResumeNext(Observable.just("7", "8", "9"));
    }
    onExceptionResumeObserver(true).subscribe(new Subscriber<String>() {
        @Override
        public void onCompleted() {
            log("onException-true-onCompleted\n");
        }
        @Override
```

```
    public void onError(Throwable e) {
        log("onException-true-onError:" + e.getMessage());
    }
    @Override
    public void onNext(String s) {
        log("onException-true-onNext:" + s);
    }
});
onExceptionResumeObserver(false).subscribe(new Subscriber<String>() {
    @Override
    public void onCompleted() {
        log("onException-false-onCompleted\n");
    }
    @Override
    public void onError(Throwable e) {
        log("onException-false-onError:" + e.getMessage());
    }
    @Override
    public void onNext(String s) {
        log("onException-false-onNext:" + s);
    }
});
```

输出的结果如下，当错误的类型是 Exception 时，在发送完 1 和 2 之后，onExceptionResumeNext 会捕获到错误事件并且用新的 Observable 来继续发送数据 7、8、9，最后整个过程正常结束；而当错误类型不是 Exception 时，onExceptionResumeNext 将不会起任何作用，错误事件被直接分发给订阅者，整个过程非正常结束。

```
onException-true-onNext:onNext:1
onException-true-onNext:onNext:2
onException-true-onNext:7
onException-true-onNext:8
onException-true-onNext:9
onException-true-onCompleted
onException-false-onNext:onNext:1
onException-false-onNext:onNext:2
onException-false-onError:Throw error
```

2.5.4 retry

retry 操作符在发生错误时会重新进行订阅，而且可以重复多次，所以发送的数据可能会产生重复。但是有可能每次 retry 都会发生错误，从而造成不断订阅不断 retry 的死循环，这种情况下可以指定最大重复次数。如果 retry 达到了最大重复次数还有错误的话，就将错误返回给观察者，其示意图如图 2-5-4 所示。

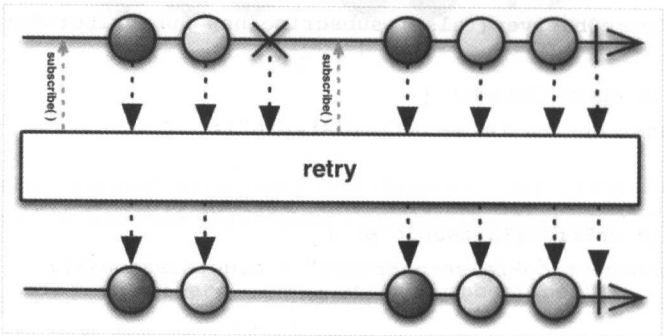


图 2-5-4

定义一个方法，创建一个会发送错误事件的 Observable，使用 retry 操作符指定最大重复次数为 2，然后进行订阅，如代码 2-5-4 所示。

代码 2-5-4

```
private Observable<Integer> createObserver() {
    return Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            log("subscribe");
            for (int i = 0; i < 3; i++) {
                if (i == 2) {
                    subscriber.onError(new Exception("Exception"));
                } else {
                    subscriber.onNext(i);
                }
            }
        }
    });
}
```

```
private Observable<Integer> retryObserver() {
    return createObserver().retry(2);
}
retryObserver().subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        log("retry-onCompleted");
    }
    @Override
    public void onError(Throwable e) {
        log("retry-onError:" + e.getMessage());
    }
    @Override
    public void onNext(Integer s) {
        log("retry-onNext:" + s);
    }
});
```

输出结果如下，Observable 每次发送出两个数据后就会发送错误事件，从而导致重新订阅，在重新订阅了两次后，错误事件最终被分发给了订阅者。

```
Subscribe
retry-onNext:0
retry-onNext:1
subscribe
retry-onNext:0
retry-onNext:1
subscribe
retry-onNext:0
retry-onNext:1
retry-onError:Exception-
```

2.6 辅助操作符

2.6.1 delay

顾名思义，delay 操作符就是让发送数据的时机延后一段时间，这样所有的数据都会依次延后一段时间再发送。在 RxJava 中将其实现为 delay 和 delaySubscription，二者的不同之处在于 delay 是延时数据的发送，而 delaySubscription 是延时注册 Subscriber，其示意图分别如图 2-6-1 和图 2-6-2 所示。

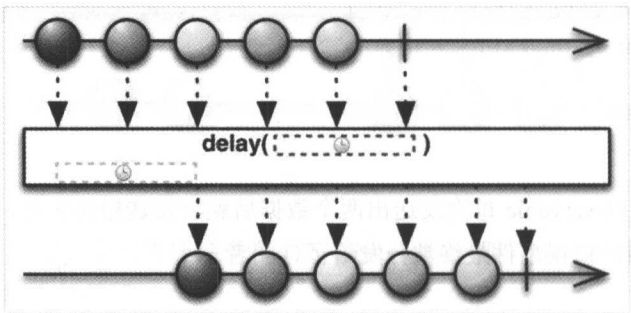


图 2-6-1

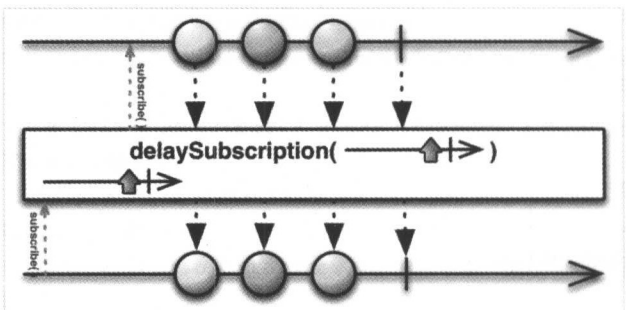


图 2-6-2

下面新建一个方法，返回一个 Observable，它会每隔一秒将当前的时间发送出来，然后使用 delay 和 delaySubscription 操作符分别延迟两秒后再进行订阅，同时将开始订阅和实际订阅的时间也都打印出来，如代码 2-6-1 所示。

代码 2-6-1

```
private Observable<Long> delayObserver() {
    return createObserver(2).delay(2000, TimeUnit.MILLISECONDS);
}
private Observable<Long> delaySubscriptionObserver() {
    return createObserver(2)
            .delaySubscription(2000, TimeUnit.MILLISECONDS);
}
private Observable<Long> createObserver(int index) {
    return Observable.create(new Observable.OnSubscribe<Long>() {
        @Override
        public void call(Subscriber<? super Long> subscriber) {
            log("subscribe:" + getCurrentTime());
            for (int i = 1; i <= index; i++) {
                subscriber.onNext(getCurrentTime());
            }
        }
    }).subscribeOn(Schedulers.newThread());
}
private long getCurrentTime() {
    return System.currentTimeMillis() / 1000;
}

log("start subscribe:" + getCurrentTime());
delayObserver().subscribe(new Action1<Long>() {
    @Override
    public void call(Long i) {
        log("delay:" + (getCurrentTime() - i));
    }
});

log("start subscribe:" + getCurrentTime());
delaySubscriptionObserver().subscribe(new Action1<Long>() {
    @Override
public void call(Long i) {
        log("delaySubscription:" + i);
    }
});
```

订阅后的输出结果如下。使用 delay 操作符后，源 Observable 收到订阅请求后立刻将数据发送了出来，但是 delay 操作符会将数据截流下来，在 2 秒后才再发送出去；而 delaySubscription 会将订阅请求截留下来，在两秒后才订阅到源 Observable 上，然后源 Observable 立刻将数据发送了出来。

```
start subscrib:1500340715
subscrib:1500340715
delay:2
delay:2
start subscrib:1500340720
subscrib:1500340722
delaySubscription:1500340722
delaySubscription:1500340722
```

2.6.2 do

do 操作符就是给 Observable 的生命周期的各个阶段加上一系列的回调监听，当 Observable 执行到这个阶段的时候，这些回调就会被触发。do 操作符在 RxJava 中有以下多个实现：

- doOnEach：Observable 每发送一个数据的时候就会触发这个回调，无论 Observable 调用的是 onNext、onError 还是 onCompleted。
- doOnNext：只有 Observable 调用 onNext 发送数据的时候才会被触发。
- doOnSubscribe 和 doOnUnsubscribe：会在 Subscriber 进行订阅和反订阅的时候触发回调。当一个 Observable 通过 OnError 或者 OnCompleted 结束的时候，会反订阅所有的 Subscriber。
- doOnError：会在 Observable 通过 OnError 分发错误事件的时候触发回调，并将 Throwable 对象作为参数传进回调函数里。
- doOnComplete：会在 Observable 通过 OnCompleted 发送结束事件的时候触发回调。
- doOnTerminate：会在 Observable 结束前触发回调，无论是正常结束还是异常结束。
- finallyDo：会在 Observable 结束后触发回调，无论是正常结束还是异常结束。

doOnEach 的示意图如图 2-6-3 所示，其他操作符的示意图在这里就不展示了，感兴趣的读

者可以到 RxJava 的官网上查看。

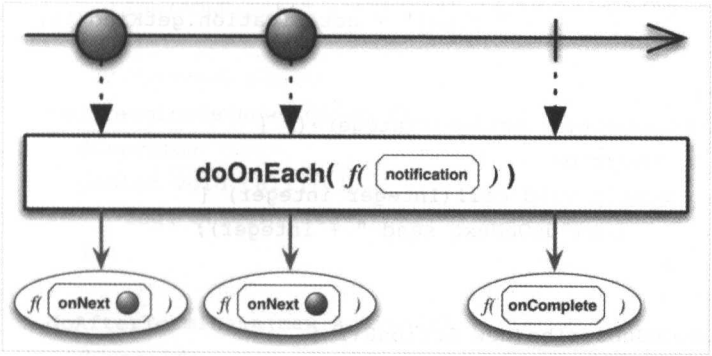


图 2-6-3

接下来我们创建两个 Observable 对象，并分别用上面的一系列 do 操作符进行注册回调。为了触发 doOnError，其中一个 Observable 对象会在数据小于等于 3 的时候发送正常的数字，在数据大于 4 的时候发送错误事件，如代码 2-6-2 所示。

代码 2-6-2

```
private Observable<Integer> createObserver() {
    return Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            for (int i = 1; i <= 5; i++) {
                if (i <= 3) {
                    subscriber.onNext(i);
                } else {
                    subscriber.onError(new Throwable("num>3"));
                }
            }
        }
    });
}

private Observable<Integer> doOnEachObserver() {
    return Observable.just(1, 2, 3)
            .doOnEach(new Action1<Notification<? super Integer>>() {
                @Override
```

```
                public void call(Notification<? super Integer> notification) {
                    log("doOnEach send " + notification.getValue()
                            + " type:" + notification.getKind());
                }
            })
            .doOnNext(new Action1<Integer>() {
                @Override
                public void call(Integer integer) {
                    log("doOnNext send " + integer);
                }
            })
            .doOnSubscribe(new Action0() {
                @Override
                public void call() {
                    log("on subscribe");
                }
            })
            .doOnUnsubscribe(new Action0() {
                @Override
                public void call() {
                    log("on unSubscribe");
                }
            })
            .doOnCompleted(new Action0() {
                @Override
                public void call() {
                    log("onCompleted");
                }
            });
    }
    private Observable<Integer> doOnErrorObserver() {
        return createObserver()
            .doOnEach(new Action1<Notification<? super Integer>>() {
                @Override
                public void call(Notification<? super Integer> notification) {
                    log("doOnEach send " + notification.getValue()
                            + " type:" + notification.getKind());
                }
            })
            .doOnError(new Action1<Throwable>() {
```

```
            @Override
            public void call(Throwable throwable) {
                log("OnError:" + throwable.getMessage());
            }
        })
        .doOnTerminate(new Action0() {
            @Override
            public void call() {
                log("OnTerminate");
            }
        })
        .doAfterTerminate(new Action0() {
            @Override
            public void call() {
                log("doAfterTerminate");
            }
        });
}

doOnEachObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("do:" + i);
    }
});

doOnErrorObserver().subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        log("onCompleted");
    }
    @Override
    public void onError(Throwable e) {
        log("subscriber onError:" + e.getMessage());
    }
    @Override
    public void onNext(Integer integer) {
        log("subscriber onNext:" + integer);
    }
});
```

订阅后的输出结果如下，请对应程序和结果体会每个 do 操作符的作用。

```
on subscribe
doOnEach send 1 type:OnNext
doOnNext send 1
do:1
doOnEach send 2 type:OnNext
doOnNext send 2
do:2
doOnEach send 3 type:OnNext
doOnNext send 3
do:3
doOnEach send null type:OnCompleted
onCompleted
on unsubscribe
doOnEach send 1 type:OnNext
subscriber onNext:1
doOnEach send 2 type:OnNext
subscriber onNext:2
doOnEach send 3 type:OnNext
subscriber onNext:3
doOnEach send null type:OnError
OnError:num>3
OnTerminate
subscriber onError:num>3
doAfterTerminate
```

2.6.3 materialize 和 dematerialize

materialize 操作符将 OnNext、OnError 和 OnComplete 事件都转化为一个 Notification 对象并按照原来的顺序发送出来，其示意图如图 2-6-4 所示。此外还有 dematerialize 操作符，它会执行相反的过程，即将这些 Notification 对象重新转化成对应的 OnNext、OnError 和 OnComplete 事件。

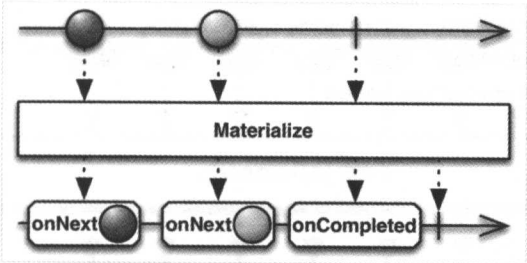


图 2-6-4

下面创建两个 Observable，使用 materialize 操作符转化一下，然后再使用 dematerialize 操作符将第二个 Observable 转化回来，如代码 2-6-3 所示。分别进行订阅查看结果。

代码 2-6-3

```
private Observable<Notification<Integer>> materializeObserver() {
    return Observable.just(1, 2, 3).materialize();
}
private Observable<Integer> deMaterializeObserver() {
    return materializeObserver().dematerialize();
}
materializeObserver().subscribe(new Action1<Notification<Integer>>() {
    @Override
    public void call(Notification<Integer> i) {
        log("materialize:" + i.getValue() + " type" + i.getKind());
    }
});
deMaterializeObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("deMeterialize:" + i);
    }
});
```

输出结果如下。可以看到 materialize 将所有的数据和事件都封装成了 Notification 对象，我们可以通过 getValue 和 getKind 方法分别将封装的值和当前的类型取出来。而 dematerialize 操作符将封装的数据又还原成了原来的数据，就如同一个普通的 Observable 一样。

```
materialize:1 typeOnNext
```

```
materialize:2 typeOnNext
materialize:3 typeOnNext
materialize:null typeOnCompleted
deMeterialize:1
deMeterialize:2
deMeterialize:3
```

2.6.4 subscribeOn 和 observeOn

这两个操作符在前面的例子中已经被使用过多次了，使用起来十分方便。在 Android 开发中，相信大家一定都遇到过不能在子线程修改 UI 的问题，所以不得不使用 Handler、AsyncTask 等来更新 UI 界面。subscribeOn 和 observeOn 操作符能让各种线程的处理都变得十分简单。

subscribeOn 用来指定 Observable 在哪个线程上运行，我们可以指定它在 IO 线程上运行，也可以让其新开一个线程运行，当然也可以在当前线程上运行。一般会指定其在各种后台线程而不是主线程上运行，就如同 AsyncTask 的 doInBackground 一样，其示意图如图 2-6-5 所示。

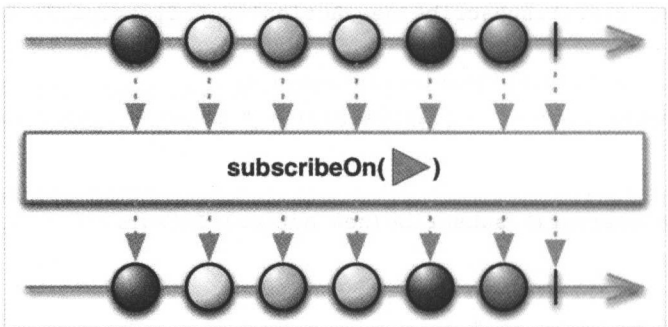


图 2-6-5

observeOn 用来指定观察者所运行的线程，也就是发送出的数据在哪个线程上使用。在 Android 中，如果我们要修改 UI 界面，观察者就必须在主线程上运行，就如同 AsyncTask 的 onPostExecute，其示意图如图 2-6-6 所示。初学者很容易混淆 subscribeOn 和 observeOn 这两个操作符，从而造成期望使用的线程弄反的问题，要怎样区分二者呢？

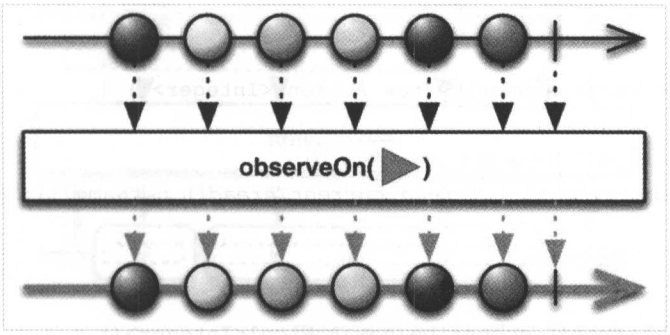


图 2-6-6

首先，subscribeOn 就是在哪个线程上订阅，订阅了 Observable 就开始干活了，所以 subscribeOn 指定的是干活的线程；然后，observeOn 就是在哪个线程上观察，观察就看结果（数据、错误和结束事件等），所以指定的就是结果被使用的线程。

下面创建两个 Observable，并使用 observeOn 和 subscribeOn 使 Observable 和观察者运行在不同的线程上，如代码 2-6-4 所示。

代码 2-6-4

```
private Observable<Integer> observeOnObserver() {
    return createObserver()
            .observeOn(AndroidSchedulers.mainThread())
            .subscribeOn(Schedulers.newThread());
}
private Observable<Integer> subscribeOnObserver() {
    return createObserver()
            .subscribeOn(Schedulers.computation())
            .observeOn(Schedulers.immediate());
}
private Observable<Integer> createObserver() {
    return Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            log("on subscribe:" + Thread.currentThread().getName());
            subscriber.onNext(1);
            subscriber.onCompleted();
        }
    });
```

```
}

observeOnObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        log("observeOn:" + Thread.currentThread().getName());
    }
});

subscribeOnObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        log("subscribeOn:" + Thread.currentThread().getName());
    }
});
```

订阅后的输出结果如下。

```
on subscribe:RxComputationScheduler-1
subscribeOn:RxComputationScheduler-1
on subscribe:RxNewThreadScheduler-1
observeOn:main
```

2.6.5 timeInterval 和 timeStamp

timeInterval 会拦截源 Observable 发送出来的数据，将其封装为一个 TimeInterval 对象，TimeInterval 对象内部包含 Observable 发送的原始数据，以及发送当前数据和发送上一个数据的时间间隔。对于第一个发送的数据，其时间间隔为订阅后到首次发送数据之间的时间间隔，其示意图如图 2-6-7 所示。

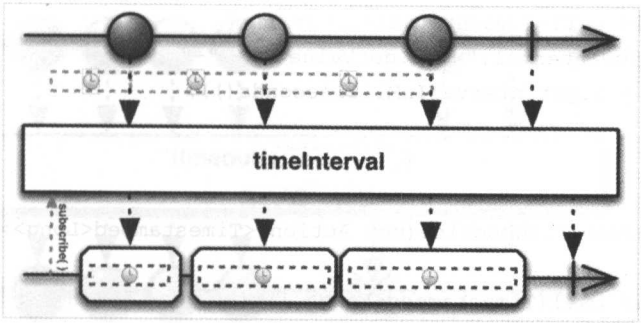


图 2-6-7

timeStamp 同样会将每个数据项重新包装成一个 TimeStamp 对象，TimeStamp 对象内包含了原始数据和一个时间戳，这个时间戳标明了每次数据发送的时间，其示意图如图 2-6-8 所示。

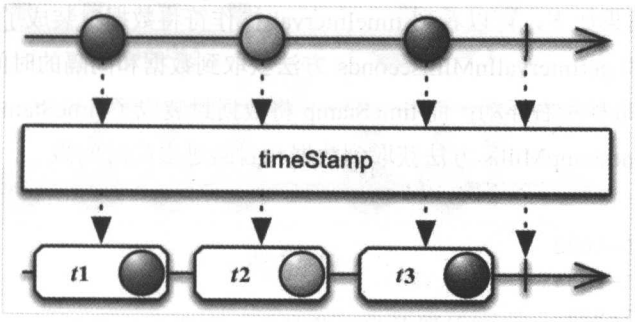


图 2-6-8

使用 interval 创建两个发送数据间隔为一秒钟的 Observable，并使用 take 操作符取 3 个数据，然后使用 timeInterval 和 timeStamp 操作符处理后进行订阅，如代码 2-6-5 所示。

代码 2-6-5

```
private Observable<TimeInterval<Long>> timeIntervalObserver() {
    return Observable.interval(1, TimeUnit.SECONDS).take(3).timeInterval();
}
private Observable<Timestamped<Long>> timeStampObserver() {
    return Observable.interval(1, TimeUnit.SECONDS).take(3).timestamp();
}

timeIntervalObserver().subscribe(new Action1<TimeInterval<Long>>() {
    @Override
```

```
        public void call(TimeInterval<Long> i) {
            log("timeInterval:" + i.getValue() + "-"
                    + i.getIntervalInMilliseconds());
        }
    });

    timeStampObserver().subscribe(new Action1<Timestamped<Long>>() {
        @Override
        public void call(Timestamped<Long> i) {
            log("timeStamp:" + i.getValue() + "-"
                    + i.getTimestampMillis());
        }
    });
```

订阅后的输出结果如下，可以看到 timeInterval 操作符将数据封装成了 timeInterval 对象，可以通过 getValue 和 getIntervalInMilliseconds 方法获取到数据和间隔的时间，由于误差存在，间隔的时间在 1000 毫秒左右浮动；而 timeStamp 将数据封装成了 timeStamped 的对象，可以通过 getValue 和 getTimestampMillis 方法获取到数据和封装进去的时间戳。

```
timeInterval:0-1002
timeInterval:1-1004
timeInterval:2-999
timeStamp:0-1500382281452
timeStamp:1-1500382282455
timeStamp:2-1500382283452
```

2.6.6 timeout

timeout 操作符给 Observable 加上超时时间，每发送一个数据后就重置计时器，当超过预定的时间还没有发送下一个数据时，就抛出一个超时的异常。

RxJava 将 timeout 实现为很多不同功能的操作符。如图 2-6-9 所示，可以在超时的时候发出一个错误事件；也可以如图 2-6-10 所示，在超时的时候使用另外一个 Observable 代替当前的 Observable 来继续发送数据。

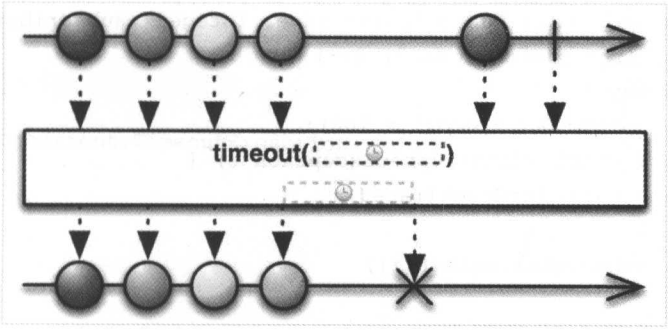


图 2-6-9

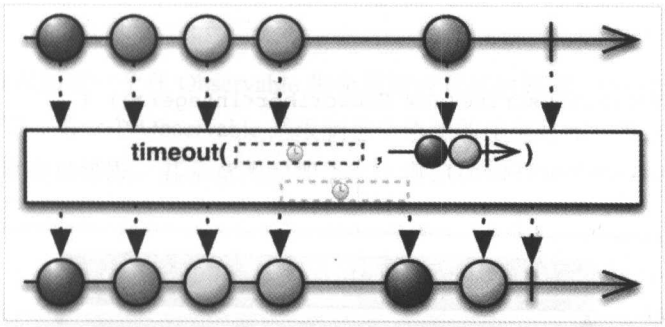


图 2-6-10

创建两个 Observable，发送 4 个数据，并随着发送数据的增加逐渐增加时间间隔，然后使用 timeout 操作符分别在超时的时候发送错误事件，并使用另外一个 Observable 来发送数据，如代码 2-6-6 所示。

代码 2-6-6

```
private Observable<Integer> timeoutObserver() {
    return createObserver().timeout(200, TimeUnit.MILLISECONDS);
}
private Observable<Integer> timeoutobserverObserver() {
    return createObserver()
            .timeout(200, TimeUnit.MILLISECONDS, Observable.just(5, 6));
}
private Observable<Integer> createObserver() {
    return Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
```

```
        public void call(Subscriber<? super Integer> subscriber) {
            for (int i = 0; i <= 3; i++) {
                try {
                    Thread.sleep(i * 100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                subscriber.onNext(i);
            }
            subscriber.onCompleted();
        }
    });
}

timeoutObserver().subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
    }
    @Override
    public void onError(Throwable e) {
        log(e);
    }
    @Override
    public void onNext(Integer integer) {
        log("timeout:" + integer);
    }
});

timeoutobserverObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        log(integer);
    }
});
```

订阅后的结果如下。可以看到第一个 Observable 发送两个数据后出现超时并发送出了错误事件；而第二个 Observable 在发送两个数据出现超时后，使用了另外的一个 Observable 来继续发送数据。

```
timeout:0
timeout:1
java.util.concurrent.TimeoutException
0
1
5
6
```

2.6.7　using

using 操作符可以创建一个在 Observable 生命周期内存活的资源，也可以这样理解：我们创建一个资源并使用它，用一个 Observable 来限制这个资源的使用时间，当这个 Observable 终止的时候，这个资源就会被销毁，其示意图如图 2-6-11 所示。

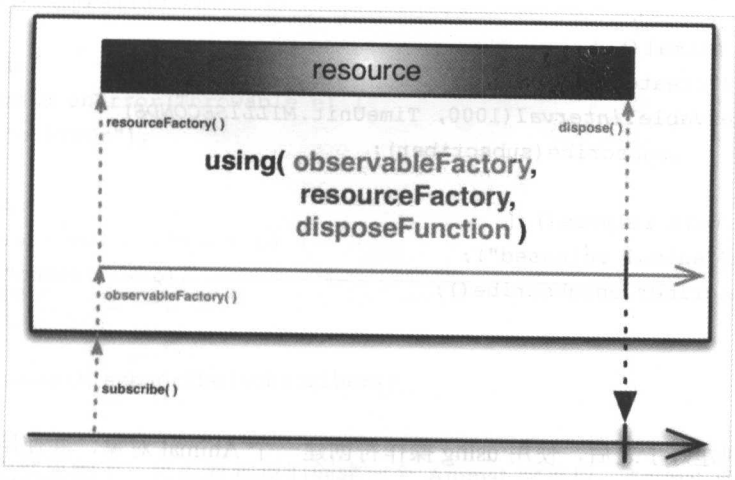


图 2-6-11

using 需要使用 3 个参数，分别是：

1. 创建这个一次性资源的函数。
2. 创建 Observable 的函数。
3. 释放资源的函数。

代码 2-6-7 定义了一个 Animal 类，在类的构造方法里我们会使用 interval 操作符创建一个 Observable 对象并对其进行订阅，然后提供一个 release 方法来反订阅这个 Observable。

代码 2-6-7

```
private class Animal {
    Subscriber subscriber = new Subscriber() {
        @Override
        public void onCompleted() {
        }
        @Override
        public void onError(Throwable e) {
        }
        @Override
        public void onNext(Object o) {
            log("animal eat");
        }
    };
    public Animal() {
        log("create animal");
        Observable.interval(1000, TimeUnit.MILLISECONDS)
                .subscribe(subscriber);
    }
    public void release() {
        log("animal released");
        subscriber.unsubscribe();
    }
}
```

Animal 类创建好了之后，使用 using 操作符创建一个 Animal 对象，并使用 timer 操作符创建一个 5 秒的控制 Observable，它在 5 秒后会调用 Animal 对象的 release 方法，也就是说这个 Animal 对象最多只能存在 5 秒，最后对其进行订阅，如代码 2-6-8 所示。

代码 2-6-8

```
private Observable<Long> usingObservable() {
    return Observable.using(new Func0<Animal>() {
        @Override
        public Animal call() {
```

```
            return new Animal();
        }
    }, new Func1<Animal, Observable<? extends Long>>() {
        @Override
        public Observable<? extends Long> call(Animal animal) {
            return Observable.timer(5000, TimeUnit.MILLISECONDS);
        }
    }, new Action1<Animal>() {
        @Override
        public void call(Animal animal) {
            animal.release();
        }
    });
}
Subscriber subscriber = new Subscriber() {
    @Override
    public void onCompleted() {
        log("onCompleted");
    }
    @Override
    public void onError(Throwable e) {
        log("onError");
    }
    @Override
    public void onNext(Object o) {
        log("onNext" + o);
    }
};
usingObservable().subscribe(subscriber);
```

订阅后的输出结果如下。可以看到我们创建一个 Animal 对象后，Animal 内部会注册一个 Observable 对象，并且每秒都会输出一个 animal eat，但是在 5 秒后会调用 Animal 的 release 方法，从而对 Animal 内部的 Observable 对象进行反订阅。

```
create animal
animal eat
animal eat
animal eat
```

```
animal eat
animal eat
onNext0
onCompleted
animal released
```

2.7 条件操作

2.7.1 all

all 操作符根据一个函数对源 Observable 发送的所有数据进行判断，最终返回的结果就是这个判断结果。这个函数使用源 Observable 发送的数据作为参数，内部判断所有的数据是否满足我们定义好的判断条件，如果全部都满足则返回 true，否则就返回 false，其示意图如图 2-7-1 所示。

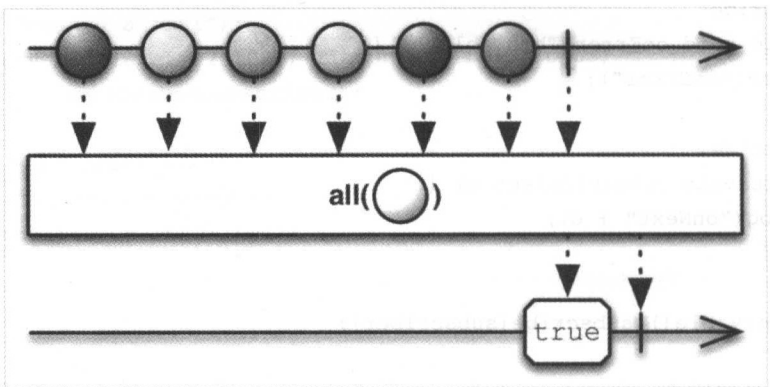


图 2-7-1

下面创建两个 Observable，一个发送 1~5 的整数，另一个发送 1~6 的整数，然后使用 all 操作符进行判断，如代码 2-7-1 所示。

代码 2-7-1

```
private Observable<Boolean> allObserver() {
    Observable<Integer> just = Observable.just(1, 2, 3, 4, 5);
```

```
        return just.all(new Func1<Integer, Boolean>() {
            @Override
            public Boolean call(Integer integer) {
                return integer < 6;
            }
        });
    }
    private Observable<Boolean> notAllObserver() {
        Observable<Integer> just = Observable.just(1, 2, 3, 4, 5, 6);
        return just.all(new Func1<Integer, Boolean>() {
            @Override
            public Boolean call(Integer integer) {
                return integer < 6;
            }
        });
    }

    allObserver().subscribe(new Action1<Boolean>() {
        @Override
        public void call(Boolean aBoolean) {
            log("all:" + aBoolean);
        }
    });
    notAllObserver().subscribe(new Action1<Boolean>() {
        @Override
        public void call(Boolean aBoolean) {
            log("not all:" + aBoolean);
        }
    });
```

订阅后的输出结果如下。可见 1~5 的整数是满足小于 6 的条件的，所以 all 操作符发送出了 true；而第二个 Observable 发送的数据中含有 6，这个数据不满足小于 6 的条件，所以 all 操作符发送出了 false。

```
all:true
not all:false
```

2.7.2 amb

amb 操作符可以将至多 9 个 Observable 结合起来，让它们竞争。哪个 Observable 首先发送了数据（包括 onError 和 onComplete），就继续发送这个 Observable 的数据，其他 Observable 所发送的数据都会被丢弃，其示意图如图 2-7-2 所示。

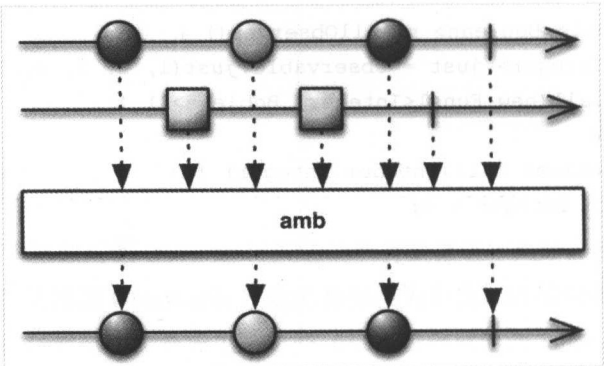


图 2-7-2

下面创建 3 个 Observable，它们在不同的时间间隔后发送不同的数据，然后使用 amb 操作符将其结合起来，如代码 2-7-2 所示。订阅查看输出。

代码 2-7-2

```
private Observable<Integer> ambObserver() {
    Observable<Integer> delay3 = Observable.just(1, 2, 3)
            .delay(3000, TimeUnit.MILLISECONDS);
    Observable<Integer> delay2 = Observable.just(4, 5, 6)
            .delay(2000, TimeUnit.MILLISECONDS);
    Observable<Integer> delay1 = Observable.just(7, 8, 9)
            .delay(1000, TimeUnit.MILLISECONDS);
    return Observable.amb(delay1, delay2, delay3);
}
ambObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("amb:" + i);
    }
});
```

订阅后的输出结果如下。由于第 3 个 Observable 最先开始发送数据，所以最终只有第 3 个 Observable 的数据发送了出来，而其他两个 Observable 的数据都被丢弃了。

```
amb:7
amb:8
amb:9
```

2.7.3　contains

contains 操作符用来判断源 Observable 所发送的所有数据是否包含某一个数据，如果包含则返回 true；如果源 Observable 已经结束了却还没有发送这个数据，则返回 false。所以在 Observable 没发送完所有的数据之前，contains 是不会有返回数据的，其示意图如图 2-7-3 所示。

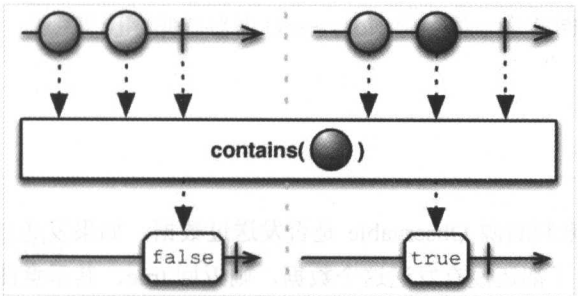


图 2-7-3

创建两个相同的 Observable，发送 1~3 的整数，然后分别用 contains 操作符判断它们所发送的数据里是否包含 3 和 4，如代码 2-7-3 所示。

代码 2-7-3

```
private Observable<Boolean> containsObserver() {
    return Observable.just(1, 2, 3).contains(3);
}
private Observable<Boolean> notContainsObserver() {
    return Observable.just(1, 2, 3).contains(4);
}
containsObserver().subscribe(new Action1<Boolean>() {
    @Override
```

```
        public void call(Boolean i) {
            log("contains:" + i);
        }
    });
    notContainsObserver().subscribe(new Action1<Boolean>() {
        @Override
        public void call(Boolean i) {
            log("not contains:" + i);
        }
    });
```

订阅后的输出结果如下。1~3 的整数当然包含 3，所以前面那个 contains 操作符最终发送出了 true；不包含 4，所以后面那个 contains 最终输出了 false。

```
contains:true
notContains:false
```

2.7.4 isEmpty

isEmpty 操作符用来判断源 Observable 是否发送过数据，如果发送过就会返回 false；如果源 Observable 已经结束了都还没有发送这个数据，则返回 true，其示意图如图 2-7-4 所示。

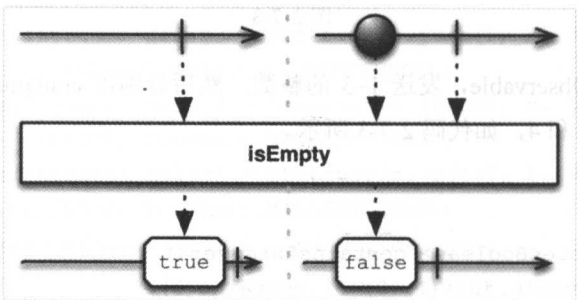


图 2-7-4

创建一个 Observable，让它在不发送任何数据的情况下结束，然后使用 isEmpty 操作符判断其是否为空，如代码 2-7-4 所示。

代码 2-7-4

```
private Observable<Boolean> emptyObserver() {
    return Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            subscriber.onCompleted();
        }
    }).isEmpty();
}
emptyObserver().subscribe(new Action1<Boolean>() {
    @Override
    public void call(Boolean i) {
        log("isEmpty:" + i);
    }
});
```

订阅后输出了 true，说明我们刚刚创建的 Observable 确实是一个空的 Observable：

```
isEmpty:true
```

2.7.5　defaultIfEmpty

defaultIfEmpty 操作符会判断源 Observable 是否发送了数据，如果源 Observable 发送了数据，则正常发送这些数据；否则发送一个默认的数据，其示意图如图 2-7-5 所示

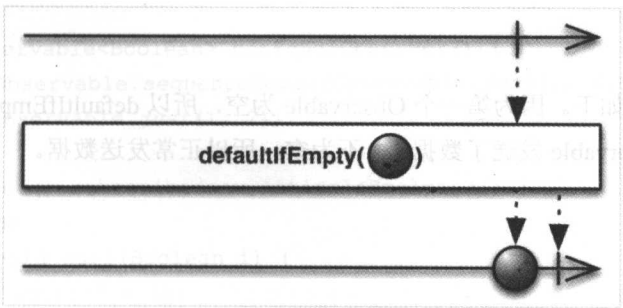


图 2-7-5

下面创建一个空的 Observable 和一个非空的 Observable，分别用 defaultIfEmpty 操作符处理，如果为空则发送出数据 10，如代码 2-7-5 所示。

代码 2-7-5

```
private Observable<Integer> emptyObserver() {
    return Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            subscriber.onCompleted();
        }
    }).defaultIfEmpty(10);
}
private Observable<Integer> notEmptyObserver() {
    return Observable.just(1).defaultIfEmpty(10);
}
emptyObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("empty:" + i);
    }
});

notEmptyObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("notEmpty:" + i);
    }
});
```

订阅后输出结果如下。因为第一个 Observable 为空，所以 defaultIfEmpty 将数据 10 发送了出来；而第二个 Observable 发送了数据 1，不为空，所以正常发送数据。

```
empty:10
notEmpty:1
```

2.7.6　sequenceEqual

sequenceEqual 操作符用来判断两个 Observable 发送的数据序列是否相同(发送的数据相同、数据的序列相同、结束的状态相同)，如果全部相同则返回 true，否则返回 false，其示意图如图 2-7-6 所示。

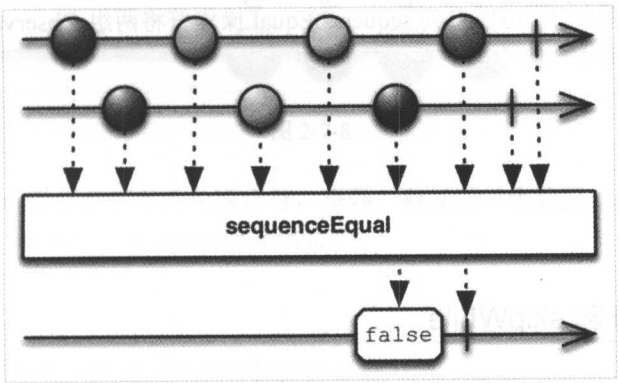


图 2-7-6

下面使用 sequenceEqual 操作符分别来比较两个发送数据相同的 Observable 和两个发送数据不同的 Observable，如代码 2-7-6 所示，然后订阅查看结果。

代码 2-7-6

```
private Observable<Boolean> equalObserver() {
    return Observable.sequenceEqual(Observable.just(1, 2, 3),
            Observable.just(1, 2, 3));
}
private Observable<Boolean> notEqualObserver() {
    return Observable.sequenceEqual(Observable.just(1, 2, 3),
            Observable.just(1, 2));
}
equalObserver().subscribe(new Action1<Boolean>() {
    @Override
    public void call(Boolean i) {
        log("equal:" + i);
    }
});
notEqualObserver().subscribe(new Action1<Boolean>() {
```

```
    @Override
    public void call(Boolean i) {
        log("notEqual:" + i);
    }
});
```

订阅后的输出结果如下。可以看到 sequenceEqual 操作符将两组 Observable 的比较结果正确地输出了出来。

```
equal:true
notEqual:false
```

2.7.7 skipUntil 和 skipWhile

这两个操作符都是根据条件来跳过一些数据，不同之处在于 skipUntil 是根据一个标志 Observable 来判断的，当这个标志 Observable 没有发送数据的时候，所有源 Observable 发送的数据都会被跳过；当标志 Observable 发送了一个数据后，则开始正常地发送数据。其示意图如图 2-7-7 所示。

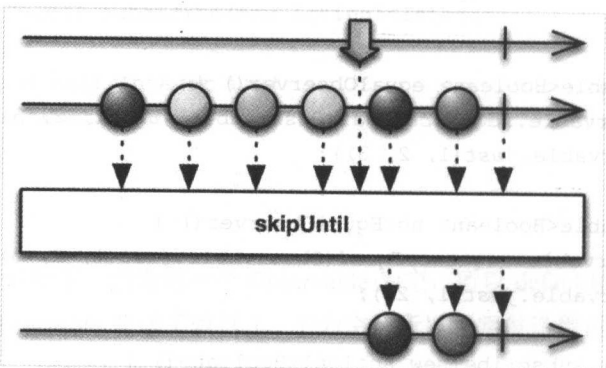


图 2-7-7

而 skipWhile 则是根据一个函数来判断是否跳过数据，如果函数返回值为 true，则一直跳过源 Observable 发送的数据；如果函数返回 false，则开始正常发送数据，其示意图如图 2-7-8 所示。

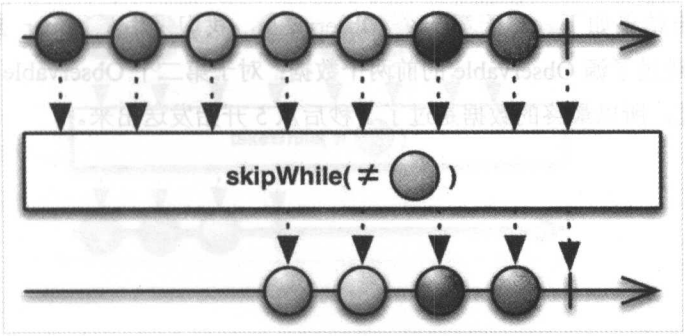


图 2-7-8

下面使用 interval 操作符创建两个操作符，每隔一秒发送一个数据。然后分别用 skipUntil 和 skipWhile 来跳过一些数据，如代码 2-7-7 所示。

代码 2-7-7

```
private Observable<Long> skipUntilObserver() {
    return Observable.interval(1, TimeUnit.SECONDS)
            .skipUntil(Observable.timer(3, TimeUnit.SECONDS));
}
private Observable<Long> skipWhileObserver() {
    return Observable.interval(1, TimeUnit.SECONDS)
            .skipWhile(aLong -> aLong < 5);
}

skipUntilObserver().subscribe(new Action1<Long>() {
    @Override
    public void call(Long i) {
        log("skipUntil:" + i);
    }
});

skipWhileObserver().subscribe(new Action1<Long>() {
    @Override
    public void call(Long i) {
        log("skipWhile:" + i);
    }
});
```

订阅后的输出结果如下。对于第一个 Observable，我们采用了 timer 操作符来创建标志 Observable，所以跳过了源 Observable 的前两个数据；对于第二个 Observable，我们的条件是小于 5 的数据都跳过，所以最终的数据是过了 5 秒后从 5 开始发送出来。

```
skipUntil:2
skipUntil:3
skipUntil:4
skipWhile:5
skipWhile:6
skipWhile:7
skipWhile:8
```

2.7.8 takeUntil 和 takeWhile

takeUntil 和 takeWhile 操作符分别和 skipUnitl 及 skipWhile 操作符是完全相反的功能。takeUntil 也是使用一个标志 Observable 是否发送数据来进行判断：当标志 Observable 没有发送数据时，正常发送数据，而一旦标志 Observable 发送过了数据，则后面的数据都会被丢弃，其示意图如图 2-7-9 所示。

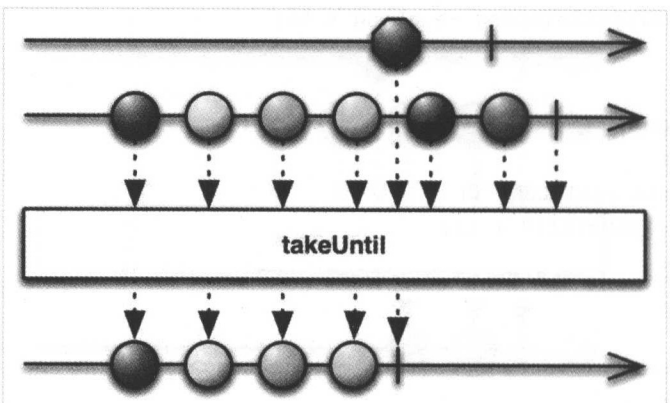


图 2-7-9

takeWhile 则是根据一个函数来判断是否发送数据，当函数返回值为 true 的时候正常发送数据；当函数返回值为 false 的时候丢弃其后面所有的数据。其示意图如图 2-7-10 所示。

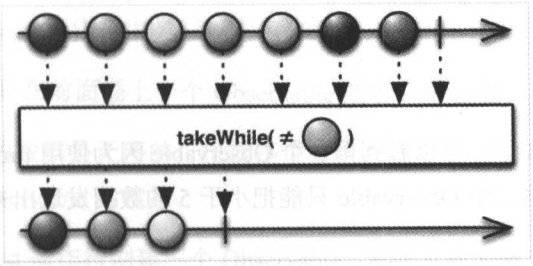


图 2-7-10

下面使用 interval 操作符创建两个 Observable，分别使用 takeUntil 和 takeWhile 操作符来取数据，如代码 2-7-8 所示。

代码 2-7-8

```
private Observable<Long> takeUntilObserver() {
    return Observable.interval(1, TimeUnit.SECONDS)
            .takeUntil(Observable.timer(3, TimeUnit.SECONDS));
}
private Observable<Long> takeWhileObserver() {
    return Observable.interval(1, TimeUnit.SECONDS)
            .takeWhile(new Func1<Long, Boolean>() {
                @Override
                public Boolean call(Long aLong) {
                    return aLong < 5;
                }
            });
}

takeUntilObserver().subscribe(new Action1<Long>() {
    @Override
    public void call(Long i) {
        log("takeUntil:" + i);
    }
});

takeWhileObserver().subscribe(new Action1<Long>() {
    @Override
    public void call(Long i) {
        log("takeWhile:" + i);
```

```
        }
    });
```

订阅后输出的结果如下。可以看到第一个 Observable 因为使用 timer 创建标志 Observable，所以只取了前两个数；第二个 Observable 只能把小于 5 的数据发送出来，大于 5 的数据都被丢弃了。

```
takeUntil:0
takeUntil:1
takeWhile:0
takeWhile:1
takeWhile:2
takeWhile:3
takeWhile:4
```

2.8 聚合操作符

2.8.1 concat

concat 操作符将多个 Observable 结合成一个 Observable 并发送数据，并且严格按照先后顺序发送数据，即前一个 Observable 的数据没有发送完时，后面的 Observable 是不能发送数据的。其示意图如图 2-8-1 所示。

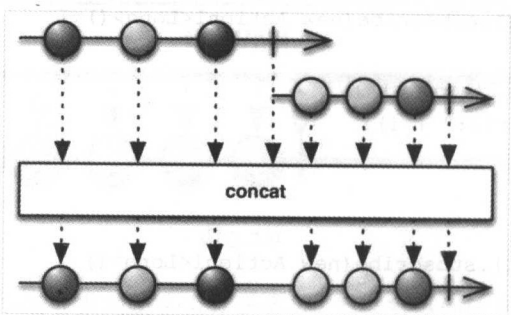


图 2-8-1

有两个操作符与 concat 操作符很类似，它们分别是：

- startWith：仅仅是在前面插上一个 Observable 或者一些数据，并且先发送插入的内容。
- merge：其发送的数据是无序的，也就是说被组合的多个 Observable 是可以自由发送数据的，而不用管其他 Observable 的状态。

代码 2-8-1 将使用 just 操作符创建三个 Observable，发送不同的数据，然后使用 concat 操作符将其组合起来并进行订阅。

代码 2-8-1

```
private Observable<Integer> concatObserver() {
    Observable<Integer> obser1 = Observable.just(1, 2, 3);
    Observable<Integer> obser2 = Observable.just(4, 5, 6);
    Observable<Integer> obser3 = Observable.just(7, 8, 9);
    return Observable.concat(obser1, obser2, obser3);
}
concatObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("concat:" + i);
    }
});
```

订阅后的结果如下。可以看到，所有的 Observable 严格按照组合时的顺序来发送数据，只有前一个 Observable 发送完所有的数据时，后一个 Observable 才开始发送数据。

```
concat:1
concat:2
concat:3
concat:4
concat:5
concat:6
concat:7
concat:8
concat:9
```

2.8.2 count

count 操作符用来统计源 Observable 发送了多少个数据，最后将数目发送出来。如果源 Observable 发送错误，则会将错误直接报出来。在源 Observable 停止发送前，count 是不会发送统计数据的，其示意图如图 2-8-2 所示。

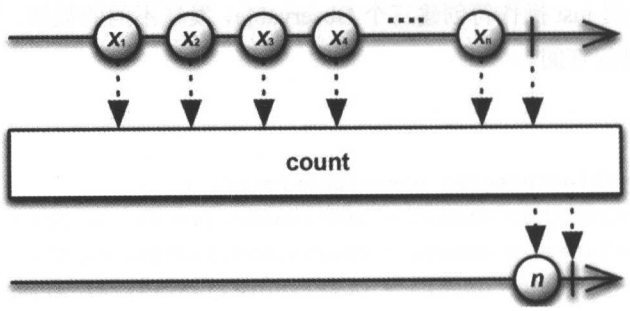


图 2-8-2

代码 2-8-2 将使用 just 操作符创建一个 Observable，然后使用 count 操作符来统计其发送的数据的数目。

代码 2-8-2

```
private Observable<Integer> countObserver() {
    return Observable.just(1, 2, 3).count();
}

countObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
        log("count:" + i);
    }
});
```

订阅后的输出结果如下。由于源 Observable 总共发送出了 3 个数据，所以我们通过 count 操作符最后得到了数量 3。

```
count:3
```

2.8.3　reduce

reduce 操作符应用一个函数接收 Observable 发送的数据和函数的计算结果，作为下次计算的参数，并输出最后的结果。reduce 与我们前面了解过的 scan 操作符很类似，只是 scan 会输出每次计算的结果，而 reduce 只输出最后的结果。reduce 的示意图如图 2-8-3 所示。

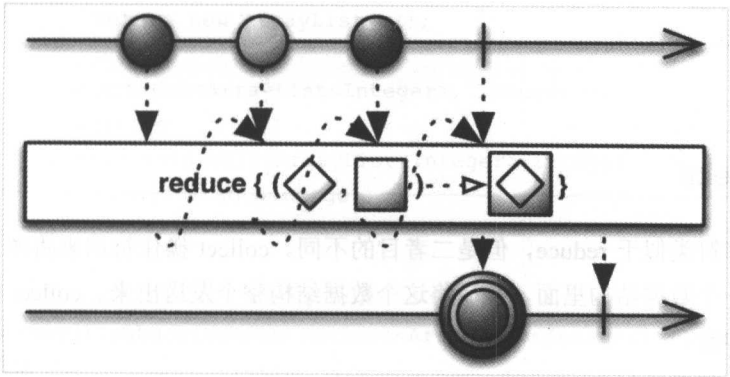


图 2-8-3

我们首先创建一个包含 10 个 2 的 list，然后使用 from 操作符以这个 list 为基础创建一个发送 10 个 2 的 Observable，并使用 reduce 操作符来计算最后的结果，如代码 2-8-3 所示。

代码 2-8-3

```
ArrayList<Integer> list = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    list.add(2);
}

return Observable.from(list).reduce((new Func2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer x, Integer y) {
        return x * y;
    }
}));

reduceObserver().subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer i) {
```

```
            log("reduce:" + i);
        }
    });
```

因为我们的 reduce 函数是相乘操作，所以最终输出的数据就是 2 的 10 次方：1024。

```
reduce:1024
```

2.8.4 collect

collect 操作符类似于 reduce，但是二者目的不同。collect 操作符用来将源 Observable 发送的数据收集到一个数据结构里面，最后将这个数据结构整个发送出来。collect 操作符需要使用两个函数作为参数：

- 第一个函数会产生收集数据结构的函数。
- 第二个函数会将上面函数产生的数据结构和源 Observable 发送的数据作为参数，且这个函数会将源 Observable 发送的数据存入到这个数据结构中。

collect 操作符的示意图如图 2-8-4 所示。

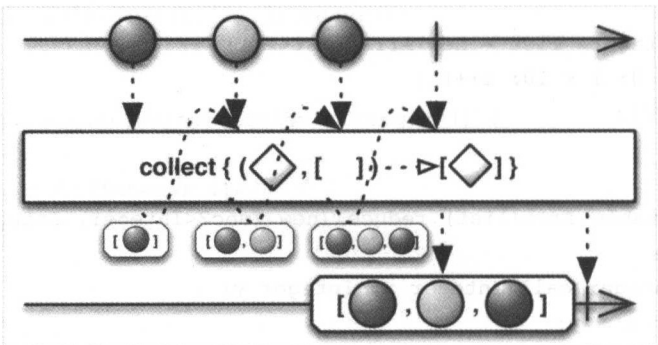


图 2-8-4

继续使用 2.8.3 节里面的 list 来创建 Observable，然后使用 collect 操作符来收集其所发送的所有数据，如代码 2-8-4 所示。

代码 2-8-4

```
private Observable<ArrayList<Integer>> collectObserver() {
    return Observable.from(list).collect(
            new Func0<ArrayList<Integer>>() {
                @Override
                public ArrayList<Integer> call() {
                    return new ArrayList<>();
                }
            }, new Action2<ArrayList<Integer>, Integer>() {
                @Override
                public void call(ArrayList<Integer> integers, Integer integer) {
                    integers.add(integer);
                }
            });
}
collectObserver().subscribe(new Action1<ArrayList<Integer>>() {
    @Override
    public void call(ArrayList<Integer> integers) {
        log("collect:" + integers);
    }
});
```

订阅后的结果如下，通过 collect 操作符，我们得到了一个打包了所有数据的 list：

```
collect:[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

2.9　与 Connectable Observable 相关的操作符

首先我们来回顾一下前面所学的 Observable，它们有一个共性，那就是只有当订阅者来订阅时才会开始发送数据，否则什么也不会发生，这就是懒加载。那什么是 Connectable Observable 呢？它是一种特殊的 Observable，并不是在订阅者订阅时才发送数据，而是只要对其应用 connect 操作符就开始发送数据。所以如果在对其应用 connect 操作符之前进行订阅的话，并不能让 Connectable Observable 发送数据。

2.9.1 publish 和 connect

publish 操作符就是用来将一个普通的 Observable 转化为一个 Connectable Observable 的。需要注意的是，如果发送数据已经开始了再进行订阅的话，就只能接收以后发送的数据。其示意图如图 2-9-1 所示。

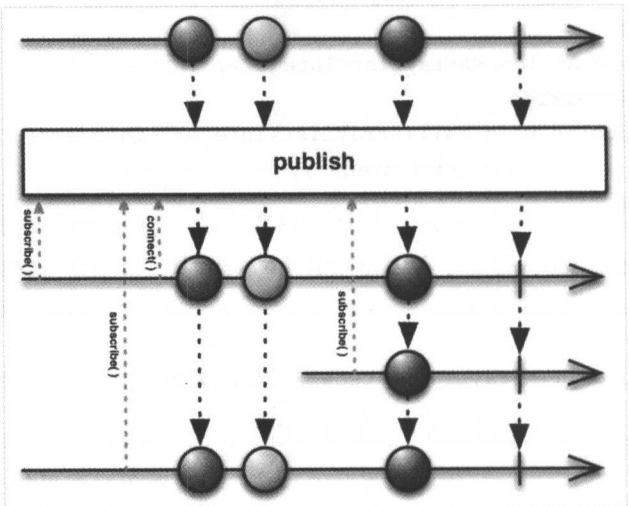


图 2-9-1

connect 操作符就是用来触发 Connectable Observable 发送数据的。应用 connect 操作符后会返回一个 Subscription 对象，通过这个 Subscription 对象，我们可以调用其 unsubscribe 方法来中止数据的发送。另外，如果还没有订阅者订阅就应用 connect 操作符，也是可以使其开始发送数据的。

下面使用 interval 操作符创建一个 Observable，它每隔一秒发送一个数据，然后使用 publish 操作符将其转化为一个 Connectable Observable。然后创建两个订阅者 Action，让 Action1 订阅到前面创建的 Connectable Observable，并且监控发送的数据，当数据为 3 时把 Action2 也订阅上，如代码 2-9-1 所示。

代码 2-9-1

```
private ConnectableObservable<Long> publishObserver() {
```

```
    Observable<Long> obser = Observable.interval(1, TimeUnit.SECONDS);
    obser.observeOn(Schedulers.newThread());
    return obser.publish();
}

Action1 action2 = new Action1() {
    @Override
    public void call(Object o) {
        log("action2:" + o);
    }
};
Action1 action1 = new Action1( {
    @Override
    public void call(Object o) {
        log("action1:" + o);
        if ((long) o == 3) obs.subscribe(action2);
    }
};
obs.subscribe(action1);
```

此时，虽然我们已经订阅了，但是因为没有使用 connnect 操作符，所以订阅者是接收不到任何数据的。所以接下来我们应用 connect 操作符让 Observable 发送数据，并可以在任何时候进行反订阅来中止数据的发送过程，如代码 2-9-2 所示。

代码 2-9-2

```
Subscription mSubscription;
mSubscription = obs.connect();
    …
if (mSubscription != null) {
    mSubscription.unsubscribe();
}
```

下面是我们使用 connect 操作符后的输出结果。当订阅者 Action1 接收到数据 3 之后就会把 Action2 也订阅上，但是 Action2 会错失前面的数据，所以接下来它们会从 4 开始接收一样的数据。当接收了 6 后，我们进行了反订阅，数据发送过程结束，所有的订阅者都收不到数据了。

```
action1:0
action1:1
action1:2
action1:3
action1:4
action2:4
action1:5
action2:5
action1:6
action2:6
```

2.9.2 refCount

refCount 操作符能够将一个 Connectable Observable 对象再重新转化为一个普通的 Observable 对象，这时候如果有订阅者进行订阅将会触发数据的发送，其示意图如图 2-9-2 所示。

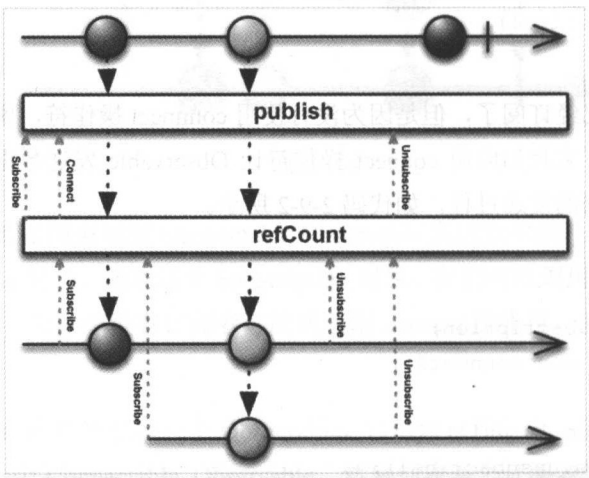


图 2-9-2

如代码 2-9-3 所示，我们首先如上文一样使用 publish 创建一个 Connectable Observable 对象，然后再使用 refCount 将其转化为一个普通的 Observable 对象，最后对其进行订阅。

代码 2-9-3

```
private ConnectableObservable<Long> publishObserver() {
```

```
    Observable<Long> obser = Observable.interval(1, TimeUnit.SECONDS);
    obser.observeOn(Schedulers.newThread());
    return obser.publish();
}

obs.refCount().subscribe(new Action1<Long>() {
    @Override
    public void call(Long aLong) {
        log("refCount:" + aLong);
    }
});
```

订阅后会让 Observable 立刻开始生产并发送数据，所以我们得到如下的输出。

```
refCount:0
refCount:1
refCount:2
...
```

2.9.3　replay

replay 操作符返回一个 Connectable Observable 对象并且可以缓存其发送过的数据，这样即使有订阅者在其发送数据之后进行订阅，也能收到其之前发送过的数据。不过使用 replay 操作符最好还是限定缓存的大小，否则如果缓存的数据太多的话，可会占用很多内存。对缓存的控制可以从空间和时间两个维度来实现，其示意图如图 2-9-3 所示。

下面我们使用 replay 来创建两个 Connectable Observable 对象，并且分别从空间和时间来控制其缓存：我们控制前一个缓存的大小为 2，控制后一个缓存的时间为 3 秒，然后如上文一样创建两个 Action，如代码 2-9-4 所示。

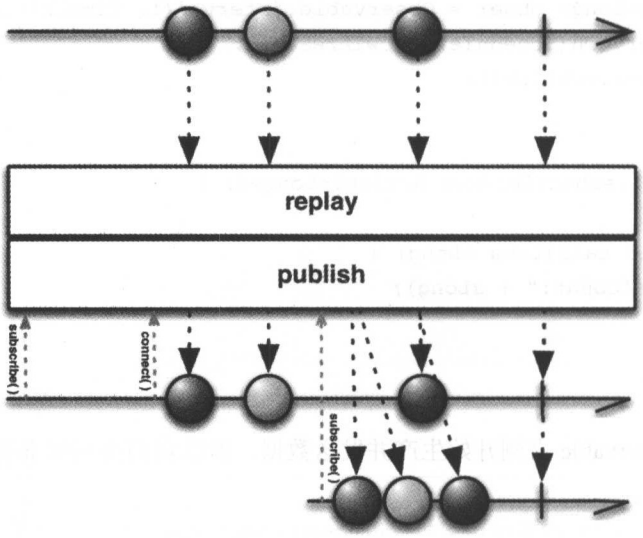


图 2-9-3

代码 2-9-4

```
private ConnectableObservable<Long> relayCountObserver() {
    Observable<Long> obser = Observable.interval(1, TimeUnit.SECONDS);
    obser.observeOn(Schedulers.newThread());
    return obser.replay(2);
}
private ConnectableObservable<Long> relayTimeObserver() {
    Observable<Long> obser = Observable.interval(1, TimeUnit.SECONDS);
    obser.observeOn(Schedulers.newThread());
    return obser.replay(3, TimeUnit.SECONDS);
}

Action1 action2 = new Action1() {
    @Override
    public void call(Object o) {
        log("action2:" + o);
    }
};
Action1 action1 = new Action1() {
    @Override
    public void call(Object o) {
```

```
        log("action1:" + o);
        if ((long) o == 3) obs.subscribe(action2);
    }
};
```

首先对控制空间缓存的 Observable 进行订阅，并使用 connect 操作符让其开始发送数据，如代码 2-9-5 所示。

代码 2-9-5

```
obs = relayCountObserver();
obs.subscribe(action1);
log("relayCount");
mSubscription = obs.connect();
```

这时将会得到如下的结果。Action1 在接收到数据 3 后把 Action2 也订阅上了，由于缓存的空间是 2，所以 Action2 可以接收到之前的两个数据 2 和 3，之后 Action1 和 Action2 会共同接收后面的数据。

```
relayCount
action1:0
action1:1
action1:2
action1:3
action2:2
action2:3
action1:4
action2:4
.....
```

下面我们订阅到控制时间缓存的 Observable，并使用 connect 操作符使其开始发送数据，如代码 2-9-6 所示。

代码 2-9-6

```
obs = relayTimeObserver();
obs.subscribe(action1);
```

```
log("relayTime");
mSubscription = obs.connect();
```

使用 connect 操作符后我们得到如下的结果。Action1 在接收到数据 3 之后把 Action2 也订阅上了，Action2 收到了之前 3 秒缓存的所有数据，之后 Action1 和 Action2 会共同接收后面的数据。

```
relayTime
action1:0
action1:1
action1:2
action1:3
action2:0
action2:1
action2:2
action2:3
action1:4
action2:4
```

2.10 自定义操作符

除了 RxJava 原生的操作符外，RxJava 还允许我们自定义操作符以满足特殊的需求。如果自定义操作符想要作用到 Observable 发送出来的数据上，就需要使用 lift 操作符；如果自定义操作符想要改变整个 Observable，就需要使用 compose 操作符，后面会详细讲述。

自定义操作符使用起来需要格外慎重，必须遵守以下几条规则。

1. 自定义操作符在发送任何数据之前都要使用 subscriber.isUnsubscribed() 来检查 Subscriber 的状态，如果没有任何 Subscriber 订阅就没有必要去发送数据了。

2. 自定义操作符要遵循 Observable 的核心原则：

   - 可以多次调用 Subscriber 的 onNext 方法，但是同一个数据只能调用一次。
   - 可以调用 Subscriber 的 onComplete 或者 onError 方法，但是这两个方法是互斥的，

调用了其中一个就不能再调用另外一个了，并且一旦调用了两者中的任何一个方法就不能再调用 onNext 方法了。

- 如果无法保证遵守以上两条原则，可以对自定义操作符加上 serialize 操作符，这个操作符会强制发送正确的数据。

3. 自定义操作符内部不能阻塞住。

4. 如果通过组合多个 RxJava 原生的操作符就能达到目的，就不要使用自定义操作符来实现，在 RxJava 的源码中就有很多这样的例子，如：

- first()操作符是通 take(1).single()来实现的。
- ignoreElements()是通过 filter(alwaysFalse())来实现的。
- reduce(a)是通过 scan(a).last()来实现的。

5. 当有异常时候，不能继续发送正常的数据，要立刻调用 Subscriber 的 onError 方法将异常抛出去。

6. 注意发送数据为 null 的情况，null 也属于一种数据，可以正常发送出去，和完全不发送数据不是一回事。

2.10.1 lift

Lift 操作符可以让我们实现一个自定义的操作符。首先我们需要创建一个实现了Operator接口的对象，在这个对象内实现我们想要自定义的操作，然后使用lift操作符可以将我们自定义的操作符和其他操作符一起做链式调用，就好像RxJava原生的操作符一样。下面我们自定义一个操作符my Operator，在这个自定义操作符里将发送的数据加上一个字符串前缀，然后使用lift操作符将这个自定义的操作符夹在两个map操作符之间，形成链式的调用，如代码 2-10-1 所示。

代码 2-10-1

```
private Observable<String> liftObserver() {
    Operator<String, String> myOperator = new Operator<String, String>() {
        @Override
        public Subscriber<? super String> call(
```

```
        Subscriber<? super String> subscriber) {
            return new Subscriber<String>(subscriber) {
                @Override
                public void onCompleted() {
                    if (!subscriber.isUnsubscribed()) {
                        subscriber.onCompleted();
                    }
                }
                @Override
                public void onError(Throwable e) {
                    if (!subscriber.isUnsubscribed()) {
                        subscriber.onError(e);
                    }
                }

            @Override
                public void onNext(String s) {
                    if (!subscriber.isUnsubscribed()) {
                        subscriber.onNext("myOperator:" + s);
                    }
                }
            };
        }
    };
    return Observable.just(1, 2, 3)
            .map(new Func1<Integer, String>() {
                @Override
                public String call(Integer integer) {
                    return "map1:" + integer;
                }
            })
            .lift(myOperator)
            .map(new Func1<String, String>() {
                @Override
                public String call(String integer) {
                    return "map2:" + integer;
                }
            });
}
```

```
liftObserver().subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
        log("lift:" + s);
    }
});
```

订阅后的输出结果如下。可以看到输出的结果也正如预期的那样，我们自定义的操作符作用在两个 map 操作符之间：

```
lift:map2:myOperator:map1:1
lift:map2:myOperator:map1:2
lift:map2:myOperator:map1:3
```

2.10.2　compose

compose 操作符需要创建一个实现了 Transformer 接口的对象。compose 操作符使用 Transformer 对象将源 Observable 按照自定义的方式转化成另外一个新的 Observable，所以也可以同其他的操作符一样形成链式调用。compose 和 lift 的区别是，compose 是对 Observable 进行操作的，而 lift 是对 Subscriber 进行操作的，也就是说它们的作用点是不同的。

下面我们自定义一个 Transformer 对象，并使用 compoe 操作符将其应用到 Observable 上。在这个自定义的 Transformer 对象中，我们将原先发送 Integer 的 Observable 转化成了一个发送 String 内容的 Observable，并且我们还添加了 doOnNext 操作符来打印出发送的数据，以方便进行后续的调试。个人感觉 Transformer 更像是一个批量转化器，如你有很多 Observable 对象在使用，可以定义一个通用的 Transformer 对象，里面可以通过 doOnNext 打印数据，可以定义 subscribeOn 和 observeOn 的线程，等等，最后使用 compose 操作符将其应用到所有的 Observable 对象上就可以统一进行设定了，如代码 2-10-2 所示。

代码 2-10-2

```
private Observable<String> composeObserver() {
    Transformer<Integer, String> myTransformer =
            new Transformer<Integer, String>() {
```

```
            @Override
            public Observable<String> call(
                    Observable<Integer> integerObservable) {
                return integerObservable
                        .map(integer -> "myTransforer:" + integer)
                        .doOnNext(s -> log("doOnNext:" + s));
            }
        };
    return Observable.just(1, 2, 3).compose(myTransformer);
}

composeObserver().subscribe(new Action1<String>() {
    @Override
    public void call(String s) {
        log("compose:" + s);
    }
});
```

订阅后的结果如下，我们可以看到每个数据都被转化为了 String，然后通过 doOnNext 输出了数据的值，之后输出了 Subscriber 接收到的数据。

```
doOnNext:myTransforer:1
compose:myTransforer:1
doOnNext:myTransforer:2
compose:myTransforer:2
doOnNext:myTransforer:3
compose:myTransforer:3
```

至此，RxJava 中的大部分操作符都讲解完了，同时给出了代码实例和运行结果，相信读者已经理解了这些操作符的作用。但是仅仅理解还是不够的，需要结合实际需求来应用合适的操作符。在第 5 章中我们将给出几个 RxJava 的应用实例来加深对操作符的理解，希望可以给读者启发。

第 3 章

使用 Scheduler 进行线程调度

在第 1 章中我们了解了 Scheduler 是 RxJava 的主要组成部分之一。在本章中，我们将了解什么是 Scheduler，如何根据具体的应用场景来使用合适的 Scheduler 进行线程调度。

3.1 什么是 Scheduler

RxJava 是一种响应式编程框架，响应式编程就是围绕着异步数据流进行的编程，而 RxJava 就是使用 Scheduler 来实现异步的。Observable 默认是单线程的，而且大部分的 Observable 默认工作在 Subscriber 调用 subscribe 方法进行订阅时所在的线程中。但是我们往往不希望 Observable 和 Subscriber 工作在同一个线程中，特别是对响应速度要求很高的 UI 线程，所以理想的情况就是耗时的 Observable 工作在一个后台线程上，当工作完成后在 UI 线程上通知 Subscriber 来更新 UI。Scheduler 可以帮我们很容易地实现异步线程调度。RxJava 使用 subscribeOn 和 observeOn 将 Scheduler 应用到 Observable 和 Subscriber 上，并且提供了多种不同的 Scheduler，分别是 io、immediate、trampoline、computation 等。此外，RxJava 还提供了一个 from 方法，可以根据传入的 Executor 对象来创建 Scheduler。除了 subscribeOn 和 observeOn 之外，在 RxJava 中的一些其他操作符在创建 Observable 时还可以将 Scheduler 作为参数，从而指定 Observable 默认的 Scheduler。

在 2.6.4 节中，我们已经介绍过 observeOn 和 subscribeOn 的使用。由于 observeOn 和 subscribeOn 也属于操作符，所以它们可以被多次嵌入到链式操作里。那么如果多次调用了

observeOn 和 subscribeOn，到底 Observable 和 Subscriber 会工作在哪个线程里呢？

- 多次调用 observeOn 会影响从其调用位置开始的后面所有操作符和 Subscriber。所以如果我们只想更改 Subscriber 的 Scheduler，应该在所有操作符的后面使用 observeOn。
- 多次调用 subscribeOn 时，只有最上面的那个起作用，所以 subscribeOn 只调用一次就行了，多了不仅没效果还容易造成困扰。

为了验证上面所说的结论，我们写程序来实际运行一下，看看程序的运行结果是否和预期的一样。如代码 3-1-1 所示，我们创建了一个 Observable，并使用 subscribeOn 操作符来多次更改 Scheduler，预期的结果会是怎样呢？

代码 3-1-1

```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? Super Integer> subscriber) {
        log("start:" + Thread.currentThread()
                .getName());
        subscriber.onNext(1);
        subscriber.onCompleted();
    }
}).subscribeOn(Schedulers.newThread())
        .map(new Func1<Integer, Integer>() {
            @Override
            public Integer call(Integer integer) {
                log(integer + ":" + Thread.currentThread()
                        .getName());
                return integer + 1;
            }
        })
        .observeOn(Schedulers.io())
        .map(new Func1<Integer, Integer>() {
            @Override
            public Integer call(Integer integer) {
                log(integer + ":" + Thread.currentThread()
                        .getName());
                return integer + 1;
            }
```

```
})
.subscribeOn(Schedulers.computation())
.subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        log("action:" + Thread.currentThread()
                .getName());
    }
});
```

运行结果如下，因为最上面的 subscribeOn 使用的是 newThread 类型的 Scheduler，所以 create 和 map 都运行在这个 Scheduler 上；然后后面 observeOn 使用的是 io 类型的 Scheduler，所以接下来的 map 和 Subscriber 都会运行在 io 的 Scheduler 上。注意在后面还有 subscribeOn 使用 computation 类型的 Scheduler，但是因为上面已经有了 subscribeOn 操作符，所以 computation 类型的 Scheduler 并没有起作用。

```
Start:RxNewThreadScheduler-1
1:RxNewThreadScheduler-1
2:RxIoScheduler-2
action:RxIoScheduler-2
```

3.2　Scheduler 的类型

在前文中，我们已经了解到 Scheduler 有 computation、io、newThread 等类型。但是不同类型的 Scheduler 有什么区别？我们又要如何选择呢？接下来我们逐个分析 RxJava 内不同类型的 Scheduler。

3.2.1　computation

computation Scheduler 适用于和 CPU 有关的任务，但是不适合那些会造成阻塞的任务，如访问磁盘和网络等。这是因为 computation Scheduler 内部会根据当前运行环境的 CPU 核心数来创建一个线程池，里面的每个线程会占用一个 CPU 的核心，从而可以充分地利用 CPU 的计算

资源。如果在 computation Scheduler 上进行阻塞的操作，当前的 Scheduler 在阻塞的时候还会占用 CPU，从而造成资源的浪费。另外，由于 CPU 的核心数是有限的，所以 computation Scheduler 内的线程也是有限的，如果有超出 CPU 核心数的任务要进行，后来的任务就必须排队等待。所以在使用 computation Scheduler 时，我们最好也能保证同时进行的任务数量小于 CPU 的核心数，这样新创建的任务会立刻申请到资源并且开始运行。

当没有使用 Scheduler 的时候，有很多和时间有关的操作符，如 delay、timer、skip、take 等，其所创建的 Observable 默认就是运行在 computation Scheduler 上的。

3.2.2 newThread

newThread Scheduler 每次都会新建一个线程。一般情况下不是很推荐使用这个 Scheduler，这是因为每次新建一个线程都会造成稍微的延迟，而且这个线程在任务结束的时候就会终结，所以也不能重用。newThread Scheduler 适合那些工作时间长并且总数少的任务，大多数情况下都可以使用 io Scheduler 来代替 newThread Scheduler。

3.2.3 io

io Scheduler 类似于 newThread Scheduler，不同之处是 io Scheduler 的线程可以被回收利用。io Scheduler 内部也会维持一个线程池，当使用 io Scheduler 来处理任务的时候，会首先从线程池中查找空闲的线程，如果有空闲线程就会在这个空闲线程上执行任务；否则就会创建一个新的线程来执行任务，并在任务执行完毕时将这个空闲的线程加入到线程池中。当然空闲的线程不会一直在那里等待，RxJava 默认空闲线程的存活时间是 60 秒，空闲时间超过 60 秒的线程会被回收。

io Scheduler 特别适合那些使用很少 CPU 资源的 I/O 操作。因为 I/O 操作一般都会花费比较长的时间来等待网络请求或者读取磁盘的返回结果，所以使用一个较大的线程池会比较合适，这样新来的任务就不需要排队等待。io Scheduler 所使用的线程池是不限大小的，所以如果有足够多的任务同时使用 io Scheduler 就会导致内存不足（OOM）的错误。

3.2.4　immediate

immediate Scheduler 会在当前的线程上立即开始执行任务，这会将当前线程上正在进行的任务阻塞。如果用 Outer 代表当前线程上正在执行的任务，用 Inner 来代表使用 immediate Scheduler 的任务，它们的执行顺序会如下所示，很明显这是一种“插队”的策略。一般来说，应该避免使用这个 Scheduler，原因会在下文中说明。

```
Outer start
Inner start
Inner end
Outer end
```

3.2.5　trampoline

trampoline Scheduler 同 immediate Scheduler 很像，都会在当前线程上执行任务。但是 trampoline 并不是立即开始执行任务的，而是等待当前线程上之前的任务都结束之后再开始执行。同样使用 Outer 和 Inner 来分别代表当前线程上的任务和使用 trampoline Scheduler 的任务，它们的执行顺序如下所示。

```
Outer start
Outer end
Inner start
Inner end
```

3.2.6　from

RxJava 内置的各种 Scheduler 可以满足绝大部分使用需求，但是不排除有一些特殊的需要无法被满足，这时我们可以使用 Schedulers.from(Executor executor)工厂方法来根据我们提供的 Executor 创建 Scheduler。如代码 3-2-1 所示，首先创建一个类实现 ThreadFactory 接口；然后新建一个 Executor 对象，设置核心和最大线程池大小为 2，将空闲线程的存活时间设置为 2 秒，使用 LinkedBlockingQueue 来作为任务排队序列，使用新建的 ThreadFactory 来创建新的线程；

最后根据我们创建的 Executor 对象获取一个 Scheduler 对象。有了这个 Scheduler 对象，我们就可以和使用其他的 Scheduler 一样来使用它了。

代码 3-2-1

```
class SimpleThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        return new Thread(r);
    }
}
Executor executor = new ThreadPoolExecutor(
        2,  //corePoolSize
        2,  //maximumPoolSize
        2000L, TimeUnit.MILLISECONDS, //keepAliveTime, unit
        new LinkedBlockingQueue<>(1000),  //workQueue
        new SimpleThreadFactory()
);
Scheduler scheduler = Schedulers.from(executor);
Observable.interval(1, TimeUnit.SECONDS).take(5)
        .observeOn(scheduler)
        .subscribe(new Action1<Long>() {
            @Override
            public void call(Long l) {
                log(l + "-"
                        + Thread.currentThread().getName());
            }
        });
```

程序运行后的结果如下，可见这个 Scheduler 对象在内部创建了两个线程，并且这两个线程会被回收利用。

```
0-Thread-10623
1-Thread-10618
2-Thread-10623
3-Thread-10618
4-Thread-10623
```

3.3　总结

至此我们已经了解了 RxJava 中不同种类的 Scheduler 及其使用场景。在实际的开发工作中，应该根据实际使用场景和各种 Scheduler 的特性来选择合适的 Scheduler。在使用 subscribeOn 和 observeOn 时要避免多次调用造成的线程调度混乱，所以我们最好在所有操作符的后面使用 subscribeOn 和 observeOn 设置对应的工作线程和观察线程。

第4章
RxJava 的实现原理

在前面几章中，我们已经了解了 RxJava 的使用方式及其强大的功能，但是到现在为止我们并不了解 RxJava 是如何工作的。可能会有很多读者对于隐藏在 RxJava 内部的原理感到好奇，在本章中，我们将深入 RxJava 的源码来探究 RxJava 究竟是如何做到数据的发送和接收，如何通过操作符来对数据做各种转换、过滤和聚合等操作，如何通过 Scheduler 来实现线程的切换等功能的。我们研究的源码将基于版本 RxJava 1.3.0。由于源码中有大量的状态检查等相关代码，为了减少篇幅，在本章中会把不太重要的代码都删掉，只关注最核心的代码。如果读者想要看完整代码，可以从官网上直接下载源码。

4.1 数据的发送和接收

通过前面的学习我们已经知道，只要创建一个 Observable，然后将一个 Subscriber 订阅到这个 Observable 上就会接收到数据，如代码 4-1-1 所示，我们通过 create 操作符创建一个 Observable，然后创建一个 Subscriber 并进行订阅，Observable 就会将数据 1 发送到 Subscriber。我们以这个例子为入口进入 RxJava 的源码来理解其数据的发送和接收机制。

代码 4-1-1

```
Observable.create(new OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        if (!subscriber.isUnsubscribed()) {
```

```
            subscriber.onNext(1);
            subscriber.onCompleted();
        }
    }
}).subscribe(new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
    }
    @Override
    public void onError(Throwable e) {
    }
    @Override
    public void onNext(Integer integer) {
        System.out.print("onNext:" + integer);
    }
});
```

4.1.1　创建 Observable 的过程

首先我们来看一下 Observable 是如何创建对象的。我们调用静态方法 create 并将创建的一个 OnSubscribe 对象作为参数传进去。如代码 4-1-2 所示，这里用到了 RxJavaHooks，RxJavaHooks 用于在代码中插入一些桩，可以监控程序的执行情况，但是大部分情况下对传入的参数都没做什么操作就返回了，所以在阅读源码的时候可以先将其忽略。创建 Observable 时，我们传入的 OnSubscribe 对象在 RxJavaHooks.onCreate 内转了一圈后就作为返回值返回，然后使用 new 创建了一个 Observable 对象。

代码 4-1-2

```
public static <T> Observable<T> create(OnSubscribe<T> f) {
    return new Observable<T>(RxJavaHooks.onCreate(f));
}
```

通过 new 创建 Observable 对象的时候自然会调用 Observable 的构造方法，如代码 4-1-3 所示，构造方法内会将传入的 OnSubscribe 参数赋值给自己的成员变量。看到这里我们就明白了，原来 Observable 就是在 OnSubscribe 外面包了一层，如图 4-1-1 所示，而创建一个 Observable 对

象的过程其实就是创建其 onSubscribe 成员变量的过程。请大家牢记 Observable 的结构和创建的过程，在阅读后面源码的过程中，我们会看到 Observable 对象被多次重复地创建。

代码 4-1-3

```
protected Observable(OnSubscribe<T> f) {
    this.onSubscribe = f;
}
```

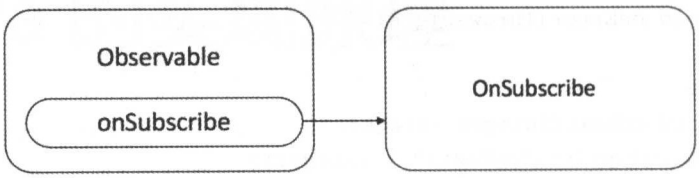


图 4-1-1

4.1.2 订阅的过程

接下来我们看一下订阅的过程，即调用了 Observable 的 subscribe 方法后到底都发生了什么。如代码 4-1-4 所示，原来 Observable 的 subscribe 方法又调用了另外一个静态 subscribe 方法并将 Observable 自身作为一个参数传入。

代码 4-1-4

```
public final Subscription subscribe(Subscriber<? super T> subscriber) {
    return Observable.subscribe(subscriber, this);
}
```

继续往里面追踪，我们省略一些无关紧要的条件检查和异常处理等操作，只看其核心操作，如代码 4-1-5 所示。

代码 4-1-5

```
static <T> Subscription subscribe(Subscriber<? super T> subscriber,
                                  Observable<T> observable) {
    subscriber.onStart();
    if (!(subscriber instanceof SafeSubscriber)) {
```

```
        subscriber = new SafeSubscriber<T>(subscriber);
    }
    RxJavaHooks.onObservableStart(observable, observable.onSubscribe)
            .call(subscriber);
    return RxJavaHooks.onObservableReturn(subscriber);
}
```

在代码 4-1-5 中，这个静态的 subscribe 方法中会进行一系列的操作。请注意，该代码段里没有将 subscriber 和 observable 的首字母大写，是因为它们是作为参数传入的对象实例，而非泛指 Subscriber 和 Observable。订阅的过程如下。

1. 调用 subscriber.onStart()来告诉 subscriber 其已经和当前的 Observable 连接起来了，但是还没有开始发送数据，subscriber 可以在这个方法里做一些初始化操作。

2. 检查一下 subscriber 对象是否是 SafeSubscriber，如果不是，则创建一个 SafeSubscriber 将 subscriber 包装起来。SafeSubscriber 可以确保 subscriber 符合标准，即 onNext 可以被调用零次或者多次，而 onError 和 onCompleted 只能被调用一次，并且三者是互斥的，而且在调用了 onError 或者 onCompleted 方法后就不能再调用 onNext 方法。

3. 这里又调用了 RxJavaHooks 方法。正如上文所说的那样，RxJavaHooks 一般不会对传入的参数做任何处理。observable.onSubscribe 对象在传入 RxJavaHooks.onObservableStart 方法后其实就是在 RxJavaHooks 里面转了一圈，没有做任何操作，最终被作为返回值返回。然后调用这个返回的 OnSubscribe 对象（即 observable.onSubscribe）的 call 方法并将 subscriber 对象传入。observable.onSubscribe 就是我们在创建 Observable 时所创建的，所以这里调用的其实就是这个 OnSubscribe 对象的 call 方法。在这个方法里，我们调用了 Subscriber 的 onNext 方法，发送了一个数据 1。这样就完成了 Observable 发送数据，Subscriber 接收数据的过程。

4. subscriber 在 RxJavaHooks 里面转了一圈，最后作为返回值返回。注意返回值的类型是 Subscription，而 Subscriber 实现了 Subscription 接口，所以这里返回 subscriber 是可以的。

看到这里，我们就明白了订阅的过程其实就是调用 Observable 内部的成员变量 onSubscribe 的 call 方法，将 Subscriber 作为参数传入进去，然后由 onSubscribe 对象向 Subscriber 发送数据的过程，如图 4-1-2 所示。另外，我们也可以理解 RxJava 中的一个知识点：在不使用 Scheduler 时，Observable 和 Subscriber 都会运行在订阅时的线程上。

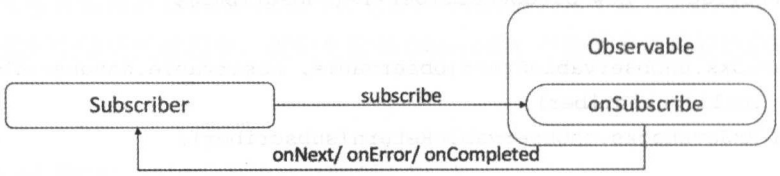


图 4-1-2

4.2 操作符的实现

在 4.1 节中，我们深入源代码了解了 Observable 的创建和订阅过程，而原始的 Observable 经过每个操作符的处理后，都会变成一个新的 Observable 对象。在本节中，我们将通过阅读一些常用操作符的源代码来理解操作符的工作方式。

4.2.1 lift 的工作原理

在 2.10.1 节中我们讲解了如何通过 lift 操作符实现自定义的操作符。lift 操作符在 RxJava 中扮演了极其重要的角色，很多操作符都是通过 lift 来实现的。下面看一下 lift 的源码部分，如代码 4-2-1 所示，lift 操作符接收一个实现了 Operator 接口的对象 operator，然后将 onSubscribe 和 operator 组合起来创建一个 OnSubscribeLift 对象，最后返回一个通过调用 unsafeCreate 创建的 Observable 对象。似乎有点复杂，让我们在下文中拆分一下每个细节。

代码 4-2-1

```
public final <R> Observable<R> lift(final Operator<? extends R,
        ? super T> operator) {
    return unsafeCreate(new OnSubscribeLift<T, R>(onSubscribe, operator));
}
```

unsafeCreate 创建的操作符需要自己实现 RxJava 的一些协议规范，我们可以先不用管其中的细节，就认为这是一个普通 create 方法，将接收一个 OnSubscribe 对象创建一个 Observable 对象（为了描述更清楚，我们称这个新建的 Observable 为 liftObservable）。

源码中创建 OnSubscribeLift 对象时传入的参数 onSubscribe 又是从哪里来的呢？lift 实际上是 Observable 类的一个方法，所以必须在一个 Observable 实例对象上才能调用 lift 方法 someObservable.lift()。还记得 4.1 节中我们分析的 Observable 的构成吗?每个 Observable 都有一个 onSubscribe 的成员变量，所以这里的这个 onSubscribe 就是源 Observable 的成员变量。

接下来看一下 OnSubscribeLift 类的实现，如代码 4-2-2 所示。同样还是去掉了异常处理等部分的相关代码，我们来看一下关键过程。

代码 4-2-2

```
public final class OnSubscribeLift<T, R> implements OnSubscribe<R> {
    final OnSubscribe<T> parent;
    final Operator<? extends R, ? super T> operator;
    public OnSubscribeLift(OnSubscribe<T> parent, Operator<? extends R,
            ? super T> operator) {
        this.parent = parent;
        this.operator = operator;
    }
    @Override
    public void call(Subscriber<? super R> o) {
        Subscriber<? super T> st = RxJavaHooks
                .onObservableLift(operator)
                .call(o);
        st.onStart();
        parent.call(st);
    }
}
```

我们结合图 4-2-1 来理解上述过程。

1. 创建一个 liftObservable，liftObservable 的 onSubscribe 成员变量即为一个 OnSubscribeLift 对象。这个 OnSubscribeLift 对象会持有一个 Observable 的 onSubscribe 成员变量的引用。

2. 当有 Subscriber 订阅的时候就会调用这个 OnSubscribeLift 对象的 call 方法。使用 RxJavaHooks 的那一段代码可以直接理解为 operator.call(o)，这样订阅时会首先调用 operator 的 call 方法返回一个包装后的 Subscriber 对象（为了描述更清楚，我们称其为 liftSubscriber）。

3. 然后再调用源 onSubscribe 的 call 方法将 liftSubscriber 传进去。

4. 源 Observable 在发送数据时，会先发送到 liftSubscriber，由 liftSubscriber 对数据做一些我们自定义的操作，然后再发送给原始的 Subscriber。

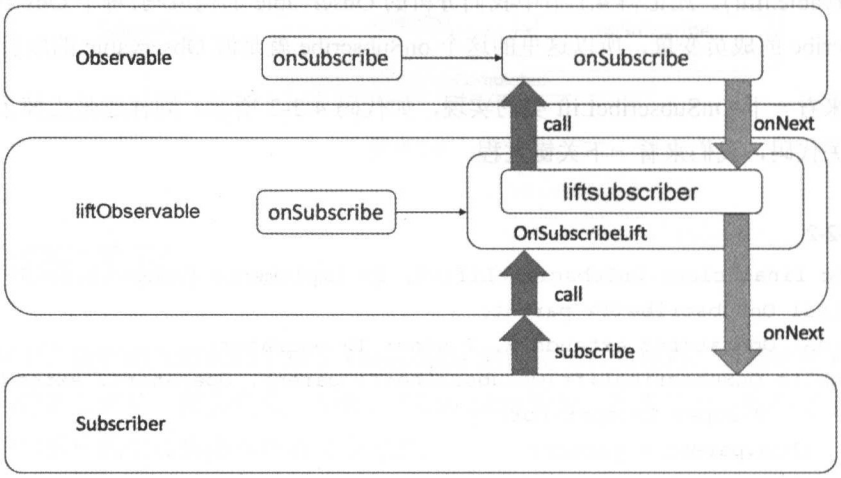


图 4-2-1

总结一下 lift 操作符的实现过程：创建一个代理 Observable 来接收源 Subscriber 的订阅，然后创建一个代理 Subscriber 订阅到源 Observable 中，在代理 Subscriber 中实现对数据的自定义操作。

4.2.2 map 的工作原理

map 操作符是一个应用非常广泛的操作符，那么它的实现原理又是怎样的呢？让我们来看一下。如代码 4-2-3 所示，map 的实现依赖于 OnSubscribeMap 类，并且会将当前的 Observable 对象和 Func1 对象一起作为构造参数，然后创建并返回一个新的 Observable，我们称之为 MapObservable。

代码 4-2-3

```
public final <R> Observable<R> map(Func1<? super T, ? extends R> func) {
    return unsafeCreate(new OnSubscribeMap<T, R>(this, func));
}
```

我们来继续深入分析，查看一下 OnSubscribeMap 的实现。如代码 4-2-4 所示，OnSubscribeMap 会将源 Observable 和 Func1 作为成员变量记录下来。由于 OnSubscribeMap 实现了 OnSubscribe 接口，所以当有 Subscriber 订阅时会调用其 call 方法。调用 call 方法时会依次做如下操作。

1. 创建一个新的 MapSubscriber：parent。

2. 将 parent 通过 add 方法添加到源 Subscriber 中，这样在取消订阅时能够将所有的 Subscriber 都取消。

3. 将 parent 订阅到源 Observable 中。

代码 4-2-4

```
public final class OnSubscribeMap<T, R> implements OnSubscribe<R> {
    final Observable<T> source;
    final Func1<? super T, ? extends R> transformer;
    public OnSubscribeMap(Observable<T> source, Func1<? super T,
            ? extends R> transformer) {
        this.source = source;
        this.transformer = transformer;
    }
    @Override
    public void call(final Subscriber<? super R> o) {
        MapSubscriber<T, R> parent = new MapSubscriber<T, R>(o, transformer);
        o.add(parent);
        source.unsafeSubscribe(parent);
    }
}
```

看到这里我们还是不清楚 map 的转换操作是怎样实现的，还需要继续深入。我们来查看一下 MapSubscriber 的实现，如代码 4-2-5 所示，同样去掉了异常处理和状态检查部分的代码，我们来关注具体实现。可以看到在 MapSubscriber 的 onNext 方法中会调用 mapper 的 call 方法，将源 Observable 发送的数据做一下转化，然后再将转化后的数据发送给源 Subscriber。其实现原理是不是和 lift 很相似呢？

代码 4-2-5

```
static final class MapSubscriber<T, R> extends Subscriber<T> {
```

```
        final Subscriber<? super R> actual;
        final Func1<? super T, ? extends R> mapper;
        public MapSubscriber(Subscriber<? super R> actual, Func1<? super T,
                ? extends R> mapper) {
            this.actual = actual;
            this.mapper = mapper;
        }
        @Override
        public void onNext(T t) {
            R result = mapper.call(t);
            actual.onNext(result);
        }
        @Override
        public void onError(Throwable e) {
            actual.onError(e);
        }
        @Override
        public void onCompleted() {
            actual.onCompleted();
        }
        @Override
        public void setProducer(Producer p) {
            actual.setProducer(p);
        }
    }
```

参考图 4-2-2，我们来总结一下 map 操作符的实现原理。

1. 创建一个 MapObservable 接受 Subscriber 的订阅。
2. MapObservable 内部创建一个 MapSubscriber 并订阅到源 Observable。
3. 源 Observable 发送数据给 MapSubscriber。
4. MapSubscriber 内部对数据按照设定的规则进行转化，并将转化后的数据发送给源 Subscriber。

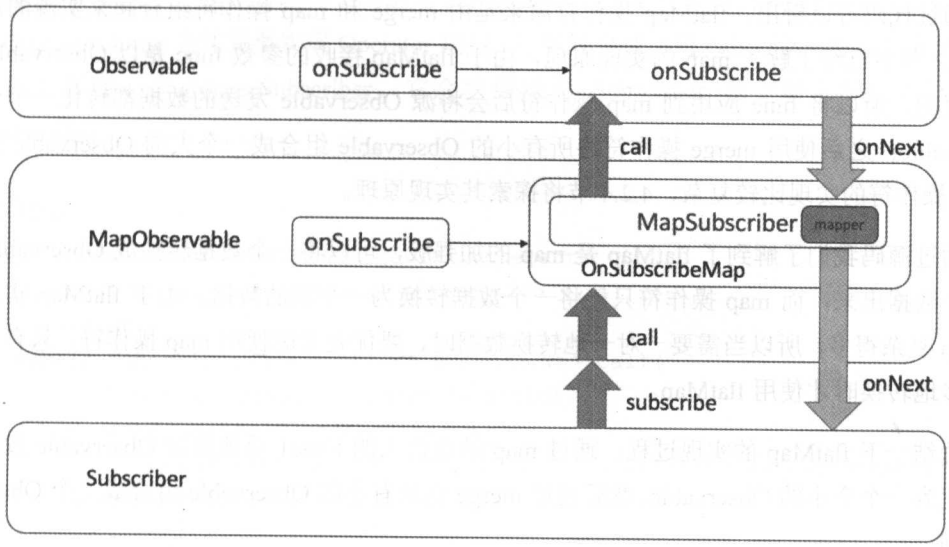


图 4-2-2

4.2.3　flatMap 的工作原理

flatMap 与 map 操作符有很多相似之处，以至于很多人都分不出二者有什么区别，通过源码我们就能很清楚地了解它们的区别了。如代码 4-2-6 所示，进入 flatMap 后首先会判断一下当前的 Observable 是否是 ScalarSynchronousObservable。当使用 just 操作符来创建 Observable 时，就会返回 ScalarSynchronousObservable，ScalarSynchronousObservable 是一种特殊的 Observable，会针对 just 的特点做一些性能的优化。在这里我们直接将其忽略，只看普通的 Observable 会走的路径，即下面的 merge(map(func))。

代码 4-2-6

```
public final <R> Observable<R> flatMap(Func1<? super T,
        ? extends Observable<? extends R>> func) {
    if (getClass() == ScalarSynchronousObservable.class) {
        return ((ScalarSynchronousObservable<T>) this).scalarFlatMap(func);
    }
    return merge(map(func));
}
```

通过代码可以看出，flatMap 操作符原来是由 merge 和 map 操作符组合起来实现的。我们在 4.2.2 节中已经了解了 map 的实现原理，由于 flatMap 接收的参数 func 是以 Observable 作为返回值的，所以将 func 应用到 map 操作符后会将源 Observable 发送的数据都转化一个个小的 Observable；之后使用 merge 操作符将所有小的 Observable 组合成一个大的 Observable 返回。merge 操作符的实现比较复杂，4.2.4 节将探索其实现原理。

通过源码我们了解到了 flatMap 是 map 的加强版，可以将一个数据转换成 Observable 后发送多个数据出来；而 map 操作符只能将一个数据转换为一个新的数据。由于 flatMap 实现起来比 map 复杂得多，所以当需要一对一地转换数据时，要优先考虑使用 map 操作符，只有当需要一对多地转换时才使用 flatMap。

总结一下 flatMap 的实现过程：通过 map 结合传入的 Func1 对象将源 Observable 发送的数据转化为一个个小的 Observable，然后通过 merge 将所有小的 Observable 组合成一个 Observable 后返回。

4.2.4 merge 的工作原理

4.2.3 节中 flatMap 的实现最终是通过 merge 来实现的，在本节中我们就来看一下 merge 的实现原理。由于 merge 支持的参数很多，所以有很多种实现，但是最终都会调用代码 4-2-7 里的这个入口。首先看其接收的参数，是一个发送小 Observable 的 Observable，请大家记住这一点，后面会用到。merge 操作符会返回一个通过 lift 转化的 Observable，我们可以称之为 MergeObservable，这里使用的 instance 传入 false 是判断是否有延迟的，可以先不考虑，直接将它看成一个 OperatorMerge 的实例。下面我们需要深入 OperatorMerge 的源码来查看其实现。

代码 4-2-7

```
public static <T> Observable<T> merge(Observable<? Extends
       Observable<? extends T>> source) {
    if (source.getClass() == ScalarSynchronousObservable.class) {
       return ((ScalarSynchronousObservable<T>) source)
              .scalarFlatMap((Func1) UtilityFunctions.identity());
    }
    return source.lift(OperatorMerge.<T>instance(false));
}
```

如代码 4-2-8 所示，我们省略掉了构造方法和 producer 等无关紧要的部分代码。为了在 lift 中使用 OperatorMerge 当然需要实现 Operator 接口，订阅时会被调用 call 方法。当 call 方法被调用时，就会新建一个 MergeSubscriber，它会将原始的 Subscriber 包一层，然后返回这个 MergeSubscriber。

代码 4-2-8

```
public final class OperatorMerge<T> implements
        Operator<T, Observable<? extends T>> {
    @Override
    public Subscriber<Observable<? extends T>> call(
            final Subscriber<? super T> child) {
        MergeSubscriber<T> subscriber =
                new MergeSubscriber<T>(child, delayErrors, maxConcurrent);
        return subscriber;
    }
}
```

继续查看 MergeSubscriber 的源码，去掉非核心代码，如代码 4-2-9 所示，可以看到其将原始的 Subscriber 作为成员变量 child 保存了起来。在本节开头时，我们已经知道了 merge 接收的参数是一个发送小 Observable 的 Observable，所以 MergeSubscriber 在 onNext 里面接收的都是这些小的 Observable。对于接收到的每个小的 Observable 都创建一个 InnerSubscriber，并对其进行订阅，这些 InnerSubscriber 会接收其订阅的 Observable 发送的数据。

代码 4-2-9

```
static final class MergeSubscriber<T> extends
        Subscriber<Observable<? extends T>> {
    final Subscriber<? super T> child;
    volatile Queue<Object> queue;
    public MergeSubscriber(Subscriber<? super T> child,
                           boolean delayErrors, int maxConcurrent) {
        this.child = child;
    }
    @Override
    public void onNext(Observable<? extends T> t) {
        InnerSubscriber<T> inner = new InnerSubscriber<T>(this, uniqueId++);
        addInner(inner);
```

```
            t.unsafeSubscribe(inner);
            emit();
        }
    }
```

让我们继续看一下 InnerSubscriber 的实现。如代码 4-2-10 所示，在构造方法里会持有一个对 MergeSubscriber 的引用，当在 onNext 中接收到数据时，就直接将其转发给外面的 MergeSubscriber。这样所有的 InnerSubscriber 接收的数据都会汇总到 MergeSubscriber 中，MergeSubscriber 再将数据转发给原始的 Subscriber 就达到 merge 的目的了。

代码 4-2-10

```
static final class InnerSubscriber<T> extends Subscriber<T> {
    final MergeSubscriber<T> parent;
    public InnerSubscriber(MergeSubscriber<T> parent, long id) {
        this.parent = parent;
        this.id = id;
    }
    @Override
    public void onStart() {
        outstanding = RxRingBuffer.SIZE;
        request(RxRingBuffer.SIZE);
    }
    @Override
    public void onNext(T t) {
        parent.tryEmit(this, t);
    }
}
```

让我们结合图 4-2-3 来总结一下 merge 的实现原理：对所有需要 merge 的 Observable 都注册一个 InnerSubscriber 来接收数据，每个 InnerSubscriber 都将接收的数据转发给外层的 MergeSubscriber，然后由 MergeSubscriber 转发给原始 Subscriber。

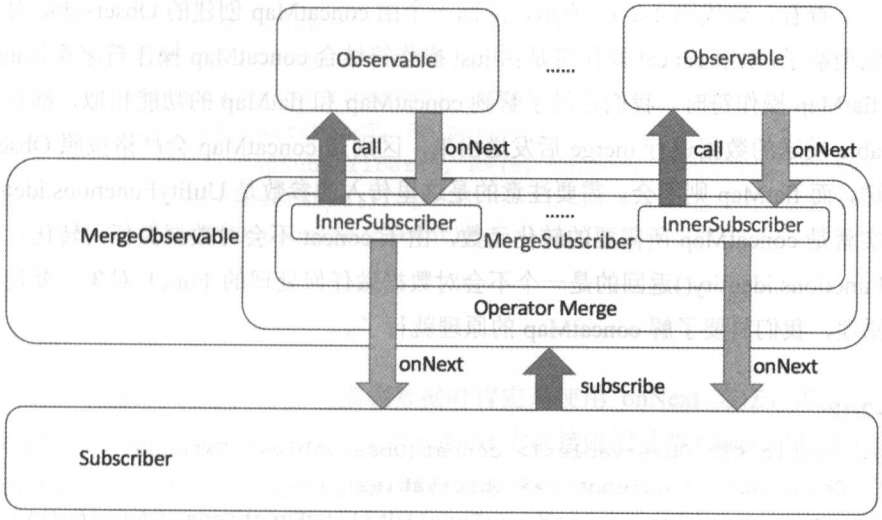


图 4-2-3

4.2.5　concat 的工作原理

在 2.8.1 节中，我们已经了解过 concat 操作符，它可以将多个 Observable 结合成一个 Observable 并发送数据，并且它会严格按照先后顺序发送数据，即前一个 Observable 的数据没有发送完时，后面的 Observable 是不能发送数据的。在第 5 章中我们将会使用 concat 结合 first 操作符来实现一个三级缓存，在这里我们就先来了解一下 concat 的实现原理。RxJava 不仅提供了可以支持 2~9 个 Observable 参数组合的多个 concat 方法，还提供了接收 Iterable 参数来支持 Observable 数目大于 9 的情况，其实现方式基本类似。在代码 4-2-11 中，我们可以看到 concat 操作符组合两个 Observable 的实现方式，原来是通过 just 操作符创建一个 Observable（不妨称之为 JustObservable）对象作为参数传入另外一个 concat 函数中。多个 Observable 的情况与之类似。

代码 4-2-11

```
public static <T> Observable<T> concat(Observable<? extends T> t1,
                                       Observable<? extends T> t2) {
    return concat(just(t1, t2));
}
```

继续往下查看，如代码 4-2-12 所示，返回一个由 concatMap 创建的 Observable 对象。看到这里我们就理解了，原来 cancat 操作符是由 just 操作符结合 concatMap 操作符来实现的。在 2.2.2 节中介绍 flatMap 操作符时，我们已经了解到 concatMap 和 flatMap 的功能相似，都是会将多个小 Observable 发送的数据进行 merge 后发送出来，区别是 concatMap 会严格按照 Observable 的顺序来发送，而 flatMap 则不会。需要注意的是这里传入的参数是 UtilityFunctions.identity()，这个参数其实就是 concatMap 所需要的转化函数，由于 concat 不会对数据做任何转化，所以这里的 UtilityFunctions.identity()返回的是一个不会对数据做任何处理的 Func1 对象。要进一步了解 concat 的原理，我们只要了解 concatMap 的原理就行了。

代码 4-2-12

```
public static <T> Observable<T> concat(Observable<? Extends
      Observable<? extends T>> observables) {
   return observables.concatMap((Func1)UtilityFunctions.identity());
}
```

接下来我们会继续忽略一些与核心原理不太相关的代码。在代码 4-2-13 中，我们看到 concatMap 返回一个 Observable（称之为 ConcatObservable），这个 Observable 的 onSubscribe 成员变量是一个新建的 OnSubscribeConcatMap 对象。

代码 4-2-13

```
public final <R> Observable<R> concatMap(Func1<? super T,
      ? extends Observable<? extends R>> func) {
   return unsafeCreate(new OnSubscribeConcatMap<T, R>(this, func,
         2, OnSubscribeConcatMap.IMMEDIATE));
}
```

当订阅时会调用 OnSubscribeConcatMap 的 call 方法。在代码 4-2-14 中，会首先将原始 Subscriber 转化为一个 SerializedSubscriber，SerializedSubscriber 可以保证在同一时刻只有一个线程能发送数据，并且 onNext、onError、onCompare 都在同一个线程中被调用，在这里我们可以先将其看作是普通的 Subscriber。SerializedSubscriber 随后会和 mapper 等参数一起被用来创建一个 ConcatMapSubscriber，并且订阅到源 Observable 上。

代码 4-2-14

```
public void call(Subscriber<? super R> child) {
    Subscriber<? super R> s;
    s = new SerializedSubscriber<R>(child);
    final ConcatMapSubscriber<T, R> parent =
            new ConcatMapSubscriber<T, R>(s, mapper, prefetch, delayErrorMode);
    if (!child.isUnsubscribed()) {
        source.unsafeSubscribe(parent);
    }
}
```

既然是一个 Subscriber，那么其接收数据时肯定是使用 onNext 方法，所以让我们直接看 ConcatMapSubscriber 的 onNext 方法。这个 onNext 方法接收的是源 Observable 发送的数据，而源 Observable 即为在本节开头处使用 just 操作符创建的 Observable，所以这里接收的数据即为想要组装的小 Observable。在 onNext 方法中，这些小 Observable 先被插入到一个队列中，然后会调用 drain 方法。

在 drain 方法中，会从队列中取出一个小的 Observable，我们都知道队列是先进先出，所以所有的小 Observable 会按照入队的顺序被取出来。取出来的小 Observable 会被 mapper 转化成新的 Observable，然后创建一个 ConcatMapInnerSubscriber 订阅到这个新的 Observable 上，此时这个新的 Observable 对象就应该开始发送数据了，如代码 4-2-15 所示。

代码 4-2-15

```
public void onNext(T t) {
    if (!queue.offer(NotificationLite.next(t))) {
        unsubscribe();
        onError(new MissingBackpressureException());
    } else {
        drain();
    }
}

void drain() {
    Object v = queue.poll();
    boolean empty = v == null;
    if (!empty) {
        Observable<? extends R> source;
```

```
            source = mapper.call(NotificationLite.<T>getValue(v));
            if (source != Observable.empty()) {
                ConcatMapInnerSubscriber<T, R> innerSubscriber =
                        new ConcatMapInnerSubscriber<T, R>(this);
                inner.set(innerSubscriber);
            }
        }
    }
```

当 ConcatMapInnerSubscriber 在 onNext 方法中接收到数据后，就会直接将数据转发到 parent 的 innerNext 方法中，而这里的 parent 即为外面的 ConcatMapSubscriber。在 ConcatMapSubscriber 的 innerNext 方法中，会将数据发送给原始的 Subscriber。当一个小的 Observable 发送完数据调用 ConcatMapInnerSubscriber 的 onCompleted 方法时，会进一步调用 ConcatMapSubscriber 的 innerCompleted 方法，而 innerCompleted 会再次调用 drain 方法，从队列中取出新的小 Observable 进行新的循环，如代码 4-2-16 所示。

代码 4-2-16

```
public void onNext(R t) {
    parent.innerNext(t);
}

public void onCompleted() {
    parent.innerCompleted(produced);
}

void innerNext(R value) {
    actual.onNext(value);
}

void innerCompleted(long produced) {
    drain();
}
```

让我们结合图 4-2-4 来总结一下 concat 的实现原理：

1. 通过 just 操作符创建一个 JustObservable，将所有需要组合的 Observable 发送出来。

2. 通过 concatMap 操作符创建一个 ConcatObservable，接受原始 Subscriber 的订阅。

3. 在 ConcatObservable 内部创建一个 ConcatMapSubscriber，并订阅到 JustObservable。

4. ConcatMapSubscriber 内部有一个队列，将 JustObservable 发送的 Observable 对象缓存起来。

5. 从队列中取出 Observable 并经过 mapper 转化（对 concat 而言并未做任何转化）。

6. 为每个从队列中取出来的 Observable 都创建一个 ConcatMapInnerSubscriber，对其进行订阅。

7. ConcatMapInnerSubscriber 将数据转发给 ConcatMapSubscriber。

8. ConcatMapSubscriber 将数据转发给原始 Subscriber。

9. 当一个小 Observable 发送完所有的数据后重复步骤 5~8，直到所有的小 Observable 都发送完数据为止。

concat 操作符之所以能保证所有的 Observable 严格地按照顺序发送数据，是因为一方面 JustObservable 会严格地按照创建时的顺序发送数据；另一方面在 ConcatMapSubscriber 内部的队列按顺序缓存了所有的小 Observable，并且只有在一个小 Observable 发送完所有的数据后，才会将下一个小 Observable 出队并继续发送数据。

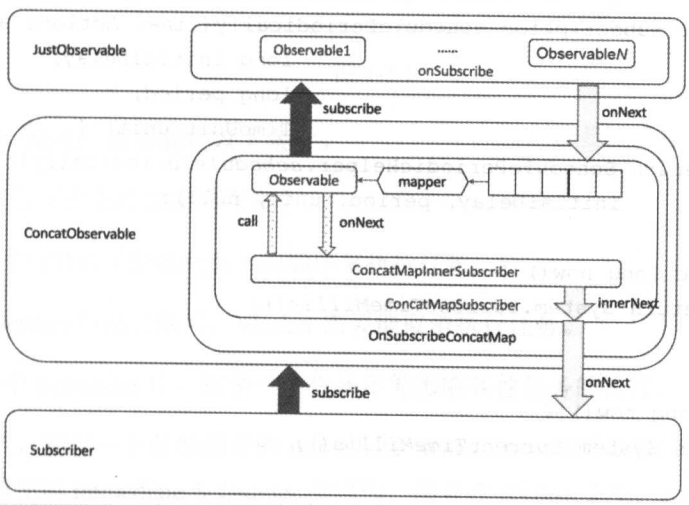


图 4-2-4

4.3 Scheduler 的工作原理

通过 subscribeOn 和 observeOn 操作符，我们能够自由地在各种不同的 Scheduler 之间切换，简直太方便了。很多人都很好奇这到底是如何做到的，在本节中，我们将通过源码来了解 Scheduler 的工作原理。首先看一下 Scheduler 类的构成。

4.3.1 Scheduler 源码

Scheduler 是一个抽象类，如代码 4-3-1 所示，其内部定义了一个抽象类 Worker，Scheduler 对外提供 createWorker 方法来创建一个 Worker 并返回。Worker 类则主要依靠 scheduler 方法来将外部将工作提交到 Worker 中。

代码 4-3-1

```
public abstract class Scheduler {
    public abstract Worker createWorker();
    public abstract static class Worker implements Subscription {
        public abstract Subscription schedule(Action0 action);
        public abstract Subscription schedule(final Action0 action,
                                              final long delayTime,
                                              final TimeUnit unit);
        public Subscription schedulePeriodically(final Action0 action,
                                                 long initialDelay,
                                                 long period,
                                                 TimeUnit unit) {
            return SchedulePeriodicHelper.schedulePeriodically(this, action,
                    initialDelay, period, unit, null);
        }
        public long now() {
            return System.currentTimeMillis();
        }
    }
    public long now() {
        return System.currentTimeMillis();
    }
    public <S extends Scheduler & Subscription> S when(
            Func1<Observable<Observable<Completable>>, Completable> combine) {
```

```
            return (S) new SchedulerWhen(combine, this);
        }
    }
```

那么，Scheduler 到底是如何实现线程调度的呢？如图 4-3-1 所示，不是 Scheduler 实现线程调度，而是当与 Scheduler 相关的操作符（observeOn、subscribeOn、timer 等）被订阅时，会调用 Scheduler 的 createWorker 方法，得到 Scheduler 在内部创建的 Worker，然后将任务传递给 Worker，Worker 可以直接在对应的线程上运行这个任务，也可以创建一些 SubWorker，然后将任务传递给 SubWorker，由 SubWorker 将任务运行在对应的线程上来完成线程调度。

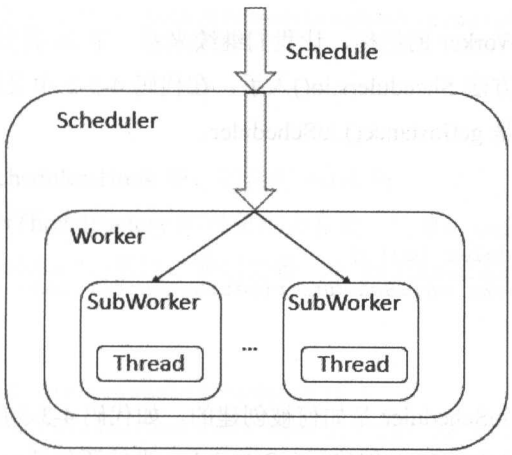


图 4-3-1

要实现一个 Worker 需要满足以下条件。

1. Worker 的所有方法都要线程安全。
2. Worker 以 FIFO（先进先出）的顺序来执行任务。
3. 当 Subscriber 取消订阅时，Worker 需要能取消执行任务。
4. 在同一个 Scheduler 中，取消一个任务不能影响其他任务的执行。

注意第 2 点，回想一下前面提及的 immediate Scheduler，很明显它并没有满足这一点，这也是我们不推荐使用 immediate Scheduler 的原因。而且在 RxJava 2 中，immediate Scheduler 已经被去掉了。

了解了 Worker 的实现要求，我们只要继承抽象类 Worker 并实现所有的虚函数，就可以创建自定义的 Worker。然后再继承抽象类 Scheduler，并实现虚函数 createWorker()，将我们之前创建的 Worker 返回就可以创建自定义 Scheduler 了。在前面的示例代码中，我们曾经多次用到过 RxAndroid 中的 AndroidSchedulers.mainThread()这个 Scheduler，这个 Scheduler 为什么可以将任务调度到主线程上呢？这是因为其内部通过 Loggper.getMainLooper()方法获得主线程的 Looper 并作为参数创建了 LooperScheduler，LooperScheduler 会根据主线程 Looper 创建一个 handler，然后以这个 handler 为参数创建了 HandlerWorker，HandlerWorker 会在这个 handler 上执行任务，从而达到了主线程执行的目的。具体的实现当然比我们这里说的要复杂得多，感兴趣的读者请自行查阅 RxAndroid 的源码。

了解了 Scheduler 和 Worker 的结构，让我们继续来看一下 io 类型的 Scheduler 是如何实现的。让我们从熟悉的静态方法 Shcedulers.io()入手，在代码 4-3-2 中又用到了 RxJavaHooks，我们将其忽略，实际返回的是 getInstance().ioScheduler。

代码 4-3-2

```
public static Scheduler io() {
    return RxJavaHooks.onIOScheduler(getInstance().ioScheduler);
}
```

那就让我们来看一下 ioScheduler 是如何被创建的。如代码 4-3-3 所示，在 Schedulers 的构造方法中通过 RxJavaSchedulersHook 创建了 ioScheduler，当然还包括一些其他类型的 Scheduler。也就说 Schedulers 类会一直持有这些 Scheduler，当需要用的时候直接拿来用即可，无须重新创建，这样可以在一定程度上提高性能。

代码 4-3-3

```
private Schedulers() {
    @SuppressWarnings("deprecation")
    RxJavaSchedulersHook hook =
            RxJavaPlugins.getInstance().getSchedulersHook();
    Scheduler c = hook.getComputationScheduler();
    if (c != null) {
        computationScheduler = c;
    } else {
        computationScheduler =
```

```
                RxJavaSchedulersHook.createComputationScheduler();
        }
        Scheduler io = hook.getIOScheduler();
        if (io != null) {
            ioScheduler = io;
        } else {
            ioScheduler = RxJavaSchedulersHook.createIoScheduler();
        }
        Scheduler nt = hook.getNewThreadScheduler();
        if (nt != null) {
            newThreadScheduler = nt;
        } else {
            newThreadScheduler = RxJavaSchedulersHook.createNewThreadScheduler();
        }
    }
```

继续进入到 RxJavaSchedulersHook 中，如代码 4-3-4 所示，这里又调用了 createIoScheduler 方法，并且创建了一个 RxThreadFactory 的对象作为参数。而 RxThreadFactory 的构造方法需要传入一个字符串"RxIoScheduler-"，所以让我们先看一下 RxThreadFactory 的实现。

代码 4-3-4

```
public static Scheduler createIoScheduler() {
    return createIoScheduler(new RxThreadFactory("RxIoScheduler-"));
}
```

在代码代码 4-3-5 中，我们可以看到 RxThreadFactory 继承了 AtomicLong，同时实现了 ThreadFactory 接口，通过 newThread 方法创建了新的线程。注意到构造方法需要传入一个字符串，在创建新的线程时，会将这个字符串作为前缀来为线程命名。而后缀则是使用 AtomicLong 的 incrementAndGet 方法得到的，这样就可以保证在创建新的线程时其名字是独一无二的，并且容易区分其类型。

代码 4-3-5

```
public final class RxThreadFactory extends
        AtomicLong implements ThreadFactory {
    private static final long serialVersionUID = -8841098858898482335L;
```

```
    public static final ThreadFactory NONE = new ThreadFactory() {
        @Override public Thread newThread(Runnable r) {
            throw new AssertionError("No threads allowed.");
        }
    };
    final String prefix;
    public RxThreadFactory(String prefix) {
        this.prefix = prefix;
    }
    @Override
    public Thread newThread(Runnable r) {
        Thread t = new Thread(r, prefix + incrementAndGet());
        t.setDaemon(true);
        return t;
    }
}
```

回到 createIoScheduler 方法，如代码 4-3-6 所示，原来返回的是一个新建的 CachedThreadScheduler。CachedThreadScheduler 类将接收 ThreadFactory 作为构造参数。

代码 4-3-6

```
public static Scheduler createIoScheduler(ThreadFactory threadFactory) {
    if (threadFactory == null) {
        throw new NullPointerException("threadFactory == null");
    }
    return new CachedThreadScheduler(threadFactory);
}
```

再进入到 CachedThreadScheduler 的构造函数，如代码 4-3-7 所示。在构造方法中将 ThreadFactory 对象保存到成员变量中，然后在 start 方法中创建了一个 CachedWorkerPool 对象并将其赋值给成员变量 pool。

代码 4-3-7

```
public CachedThreadScheduler(ThreadFactory threadFactory) {
    this.threadFactory = threadFactory;
    this.pool = new AtomicReference<CachedWorkerPool>(NONE);
```

```
        start();
    }
    @Override
    public void start() {
        CachedWorkerPool update =
            new CachedWorkerPool(threadFactory, KEEP_ALIVE_TIME, KEEP_ALIVE_UNIT);
        if (!pool.compareAndSet(NONE, update)) {
            update.shutdown();
        }
    }
```

让我们先把 CachedWorkerPool 放一放。前文说过，Scheduler 需要自己实现一个 Worker，并且在外部调用 createWorker 的时候返回一个 Worker 实例。如代码 4-3-8 所示，在调用 createWorker 时返回了一个 EventLoopWorker 对象，而 EventLoopWorker 又依赖于 pool，即 CachedWorkerPool。

代码 4-3-8

```
public Worker createWorker() {
    return new EventLoopWorker(pool.get());
}
```

让我们继续看一下 EventLoopWorker 的实现。如代码 4-3-9 所示，当外部调用 schedule 方法，将一个 Action0 对象传入的时候，会最终调用 threadWorker.scheduleActual 方法，在这个方法内调用 Action0 对象的 call 方法。

代码 4-3-9

```
static final class EventLoopWorker extends Scheduler.Worker
        implements Action0 {
    private final CachedWorkerPool pool;
    private final ThreadWorker threadWorker;
    final AtomicBoolean once;
    EventLoopWorker(CachedWorkerPool pool) {
        this.pool = pool;
        this.once = new AtomicBoolean();
        this.threadWorker = pool.get();
    }
```

```
    @Override
    public Subscription schedule(Action0 action) {
        return schedule(action, 0, null);
    }
    @Override
    public Subscription schedule(final Action0 action,
                                 long delayTime, TimeUnit unit) {
        ScheduledAction s = threadWorker.scheduleActual(new Action0() {
            @Override
            public void call() {
                if (isUnsubscribed()) {
                    return;
                }
                action.call();
            }
        }, delayTime, unit);
        return s;
    }
}
```

让我们看一下 scheduleActual 的实现。如代码 4-3-10 所示，原来提交过来的 Action0 对象会被包装成 ScheduledAction 类型，而 ScheduledAction 实现了 Runnable 接口，这样就被提交到了 executor 里面执行，而 executor 则是在调用 NewThreadWorker 时创建的一个 ScheduledExecutorService 实例。

代码 4-3-10

```
public ScheduledAction scheduleActual(final Action0 action, long delayTime,
                                      TimeUnit unit) {
    Action0 decoratedAction = RxJavaHooks.onScheduledAction(action);
    ScheduledAction run = new ScheduledAction(decoratedAction);
    Future<?> f;
    if (delayTime <= 0) {
        f = executor.submit(run);
    } else {
        f = executor.schedule(run, delayTime, unit);
    }
    run.add(f);
    return run;
}
```

```
public NewThreadWorker(ThreadFactory threadFactory) {
    ScheduledExecutorService exec = Executors.
            newScheduledThreadPool(1, threadFactory);
    executor = exec;
}
```

了解了 ThreadWorker 的工作原理，我们需要回到代码 4-3-9，EventLoopWorker 类的成员变量 threadWorker 是在构造函数里通过 pool.get() 方法得到的。所以下一步就是看一下 CachedWorkerPool 在内部到底做了什么，通过其 get 方法又会返回什么。

CachedWorkerPool 就是 io 类型 Scheduler 最关键的部分了。如代码 4-3-11 所示，在构造方法里会创建一个 ConcurrentLinkedQueue 的成员变量 expiringWorkerQueue。当调用 get 方法时，会先到 expiringWorkerQueue 里面去查找是否有缓存的 ThreadWorker 对象，如果有则会直接返回缓存的 ThreadWorker 对象，如果没有则会创建一个新的并返回。当一个任务执行完毕的时候，就会调用 release 方法，在更新了 ThreadWorker 的过期时间后将其插入到 expiringWorkerQueue 中。

代码 4-3-11

```
static final class CachedWorkerPool {
    private final ThreadFactory threadFactory;
    private final long keepAliveTime;
    private final ConcurrentLinkedQueue<ThreadWorker> expiringWorkerQueue;
    private final CompositeSubscription allWorkers;
    private final ScheduledExecutorService evictorService;
    private final Future<?> evictorTask;
    CachedWorkerPool(final ThreadFactory threadFactory,
                     long keepAliveTime, TimeUnit unit) {
        this.threadFactory = threadFactory;
        this.keepAliveTime = unit != null ? unit.toNanos(keepAliveTime) : 0L;
        this.expiringWorkerQueue = new ConcurrentLinkedQueue<ThreadWorker>();
        this.allWorkers = new CompositeSubscription();
        ScheduledExecutorService evictor = null;
    }
    ThreadWorker get() {
        if (allWorkers.isUnsubscribed()) {
            return SHUTDOWN_THREADWORKER;
        }
```

```
            while (!expiringWorkerQueue.isEmpty()) {
                ThreadWorker threadWorker = expiringWorkerQueue.poll();
                if (threadWorker != null) {
                    return threadWorker;
                }
            }
            ThreadWorker w = new ThreadWorker(threadFactory);
            allWorkers.add(w);
            return w;
        }

    void release(ThreadWorker threadWorker) {
        threadWorker.setExpirationTime(now() + keepAliveTime);
            expiringWorkerQueue.offer(threadWorker);
    }

    }
```

总结一下 io 类型的 Scheduler 实现原理：

1. Scheduler 所提供的 Worker 对象为 EventLoopWorker。

2. EventLoopWorker 有一个 ThreadWorker 类型的成员变量，由这个 ThreadWorker 成员变量来实际执行任务。

3. 这个 ThreadWorker 成员变量是从 CachedWorkerPool 获取的。

4. CachedWorkerPool 内部有一个缓存队列，使用完的 ThreadWorker 对象会在过期时间内缓存在队列中，当有新的任务请求时会优先使用缓存队列中的 ThreadWorker。

了解了 Scheduler 的结构和工作原理，我们来结合源码看看 subscribeOn 和 observerOn 是如何实现在不同 Scheduler 之间切换的。

4.3.2 subscribeOn 的工作原理

通过 subscribeOn 可以让 Observable 在指定的 Scheduler 上发送数据，即让其在这个线程上工作。在代码 4-3-12 中，我们在主线程上使用 just 创建了一个 Observable，然后用 subscribeOn 让其工作在 io 类型的 Scheduler 上，最后将一个 Subscriber 订阅上来接收数据，让我们来看一下

这段代码中 subscribeOn 是如何工作的。

代码 4-3-12

```
Observable.just(1)
        .subscribeOn(Schedulers.io())
        .subscribe(new Subscriber<Integer>() {
            @Override
            public void onCompleted() {
            }
            @Override
            public void onError(Throwable e) {
            }
            @Override
            public void onNext(Integer integer) {
                System.out.print("onNext:" + integer);
            }
        });
```

subscribeOn 内部会有一些跳转，我们将其忽略来看主要部分。如代码 4-3-13 所示，在 subscribeOn 方法内部首先会有对 ScalarSynchronousObservable 的优化，这部分也不是重点；然后可以看到通过 unsafeCreate 方法创建了一个新的 Observable 并返回，我们暂时称之为 SubcriberOnObserverable；接下来进一步深入到 OperatorSubscribeOn 中。

代码 4-3-13

```
public final Observable<T> subscribeOn(Scheduler scheduler,
                                boolean requestOn) {
    if (this instanceof ScalarSynchronousObservable) {
        return ((ScalarSynchronousObservable<T>) this)
                .scalarScheduleOn(scheduler);
    }
    return unsafeCreate(new OperatorSubscribeOn<T>(this,
            scheduler, requestOn));
}
```

如代码 4-3-14 所示，OperatorSubscribeOn 实现了 OnSubscribe 接口，也就是说上一步新建的那个 SubcriberOnObserverable 的 onSubscribe 成员变量就是 OperatorSubscribeOn 的一个实例，

当有 Subscriber 订阅时，将会调用其 call 方法。

在 call 方法中，首先从传入的 Scheduler 对象获取到 Worker 实例 inner，然后创建一个 SubscribeOnSubscriber 对象的实例 parent，将源 Subscriber 和源 Observer 都包装起来。最后调用 inner 的 scheduler 方法将 parent 传进去。

代码 4-3-14

```
public final class OperatorSubscribeOn<T> implements OnSubscribe<T> {
    final Scheduler scheduler;
    final Observable<T> source;
    final boolean requestOn;
    public OperatorSubscribeOn(Observable<T> source,
                               Scheduler scheduler, boolean requestOn) {
        this.scheduler = scheduler;
        this.source = source;
        this.requestOn = requestOn;
    }
    @Override
    public void call(final Subscriber<? super T> subscriber) {
        final Worker inner = scheduler.createWorker();
        SubscribeOnSubscriber<T> parent =
                new SubscribeOnSubscriber<T>(subscriber,
                        requestOn, inner, source);
        subscriber.add(parent);
        subscriber.add(inner);
        inner.schedule(parent);
    }
}
```

让我们继续看一下 SubscribeOnSubscriber 内部都做了什么。如代码 4-3-15 所示，可以看到其实现了 Action0 接口，这样调用 Worker 的 scheduler 方法时就会在 Worker 所在的线程上调用 SubscribeOnSubscriber 的 call 方法。而在 call 方法内，SubscribeOnSubscriber 将自己注册到源 Observable 上。在 4.1.2 节中我们已经了解到了 Observable 会工作在订阅时的线程上，这样通过 subscribeOn 就会在 Worker 的线程上进行订阅，从而 Observable 也就会在这个线程上进行工作了。由于 SubscribeOnSubscriber 会直接将数据转发给源 Subscriber，所以如果没有使用 observeOn 操作符切换 Scheduler 的话，源 Subscriber 也会在 Worker 的线程上接收到数据。

代码 4-3-15

```
static final class SubscribeOnSubscriber<T> extends
        Subscriber<T> implements Action0 {
    final Subscriber<? super T> actual;
    final boolean requestOn;
    final Worker worker;
    Observable<T> source;
    Thread t;
    SubscribeOnSubscriber(Subscriber<? super T> actual,
                          boolean requestOn, Worker worker,
                          Observable<T> source) {
        this.actual = actual;
        this.requestOn = requestOn;
        this.worker = worker;
        this.source = source;
    }
    @Override
    public void onNext(T t) {
        actual.onNext(t);
    }
    @Override
    public void onError(Throwable e) {
        try {
            actual.onError(e);
        } finally {
            worker.unsubscribe();
        }
    }
    @Override
    public void onCompleted() {
        try {
            actual.onCompleted();
        } finally {
            worker.unsubscribe();
        }
    }
    @Override
    public void call() {
        Observable<T> src = source;
        source = null;
```

```
            t = Thread.currentThread();
            src.unsafeSubscribe(this);
        }
    }
```

让我们结合图 4-3-2 来总结一下 subscribeOn 的工作原理：

1. subscribeOn 会创建并返回一个 Observable（图 4-3-1 中的 SubscriberOnObservable）。

2. 当原始 Subscriber 在订阅线程（用白色箭头表示）上订阅到 SubscriberOnObservable 时，在 SubscriberOnObservable 内会创建一个代理 Subscriber（SubscribeOnSubscriber）。

3. SubscribeOnSubscriber 会在 Scheduler 线程上（用灰色箭头表示）注册到源 Observable 上，从而源 Observable 就会工作在 Scheduler 线程上。

4. SubscribeOnSubscriber 在 Scheduler 线程上接收到从源 Observable 发送来的数据就会直接转发给原始 Subscriber。所以从数据流的角度来看，subscribeOn 会影响其上游和下游的操作。

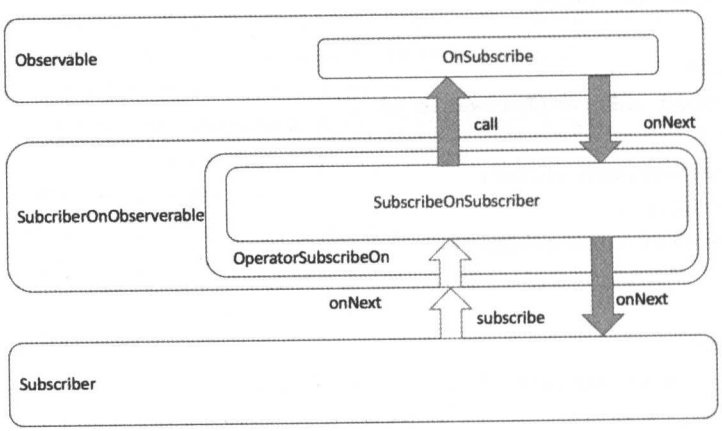


图 4-3-2

4.3.3 observeOn 的工作原理

进入 observeOn 也会进行一些方法的跳转，我们略过这些跳转直接看关键的部分。如代码 4-3-16 所示，observeOn 方法使用了 lift 操作符来返回一个新的 Observable，我们还需要进一步

查看 OperatorObserveOn 的实现。

代码 4-3-16

```
public final Observable<T> observeOn(Scheduler scheduler,
                                     boolean delayError, int bufferSize) {
    return lift(new OperatorObserveOn<T>(scheduler, delayError, bufferSize));
}
```

OperatorObserveOn 实现了 Operator 接口，这样当有 Subscriber 订阅的时候会调用其 call 方法。在 call 方法内先判断一下所使用的 Scheduler 是否是 ImmediateScheduler 或 TrampolineScheduler，如果是会直接返回。也就是说 observerOn 使用这两种 Scheduler 时是不会做任何操作的。检查通过后，创建一个 ObserveOnSubscriber 对象将使用的 Scheduler 和源 Subscriber 都封装起来，并返回这个 ObserveOnSubscriber 对象，如代码 4-3-17 所示。

代码 4-3-17

```
public final class OperatorObserveOn<T> implements Operator<T, T> {
    private final Scheduler scheduler;
    private final boolean delayError;
    private final int bufferSize;
    public OperatorObserveOn(Scheduler scheduler, boolean delayError) {
        this(scheduler, delayError, RxRingBuffer.SIZE);
    }
    public OperatorObserveOn(Scheduler scheduler,
                             boolean delayError, int bufferSize) {
        this.scheduler = scheduler;
        this.delayError = delayError;
        this.bufferSize = (bufferSize > 0) ? bufferSize : RxRingBuffer.SIZE;
}
    @Override
    public Subscriber<? super T> call(Subscriber<? super T> child) {
        if (scheduler instanceof ImmediateScheduler) {
            return child;
        } else if (scheduler instanceof TrampolineScheduler) {
            return child;
        } else {
            ObserveOnSubscriber<T> parent =
                    new ObserveOnSubscriber<T>(scheduler,
```

```
                    child, delayError, bufferSize);
            parent.init();
            return parent;
        }
    }
}
```

接下来继续看一下 ObserveOnSubscriber 的实现。这个实现较为复杂，我们删掉逻辑中其他的部分，只看数据的传递，如代码 4-3-18 所示。

1. 在构造方法内调用 scheduler.createWorker()获取一个 Work 对象，并保存在成员变量 recursiveScheduler 中，这是核心所在。

2. 当通过 onNext 方法接收到数据时，将数据插入到队列中，并调用 schedule 方法。

3. 在 schedule 方法里会调用 recursiveScheduler.schedule，将自己作为对象传进去，这样就会在这个 Worker 的线程上调用自己的 call 方法。

4. 在 call 方法内将数据从队列中取出来，并调用 localChild.onNext 将数据发送出去，只有这一步是在 Schedule 指定的线程上执行的，所以只能影响后续的操作，而不会影响前面 Observable 的工作。

代码 4-3-18

```
static final class ObserveOnSubscriber<T> extends Subscriber<T> implements
Action0 {
    final Subscriber<? super T> child;
    final Scheduler.Worker recursiveScheduler;
    final Queue<Object> queue;
    public ObserveOnSubscriber(Scheduler scheduler, Subscriber<? super T> child,
                               boolean delayError, int bufferSize) {
        this.child = child;
        this.recursiveScheduler = scheduler.createWorker();
        if (UnsafeAccess.isUnsafeAvailable()) {
            queue = new SpscArrayQueue<Object>(calculatedSize);
        } else {
            queue = new SpscAtomicArrayQueue<Object>(calculatedSize);
        }
    }
```

```
    @Override
    public void onNext(final T t) {
        if (!queue.offer(NotificationLite.next(t))) {
            onError(new MissingBackpressureException());
            return;
        }
        schedule();
    }
    protected void schedule() {
        if (counter.getAndIncrement() == 0) {
            recursiveScheduler.schedule(this);
        }
    }
    @Override
    public void call() {
        final Queue<Object> q = this.queue;
        final Subscriber<? super T> localChild = this.child;
         localChild.onNext(NotificationLite.<T>getValue(v));
    }
}
```

让我们结合图 4-3-3 总结一下 observeOn 的工作原理：

1. observeOn 创建一个 Observable（ObserveOnObserverable）。

2. 当原始 Subscriber 在订阅线程上（用白色箭头表示）订阅到 ObserveOnObserverable 时，在 ObserveOnObserverable 内会创建一个代理 Subscriber（ObserveOnSubscriber）。

3. ObserveOnSubscriber 继续在订阅线程上订阅到源 Observable，所以此时源 Observable 是工作在订阅线程上的。

4. ObserveOnSubscriber 在订阅线程上接收到数据后会先将数据缓存到队列中，然后在 Scheduler 的线程上（用灰色箭头表示）将数据分发给原始 Subscriber，从而达到在 Scheduler 的线程上接收数据的目的。所以从数据流的角度看，observeOn 只会影响其下游的操作。

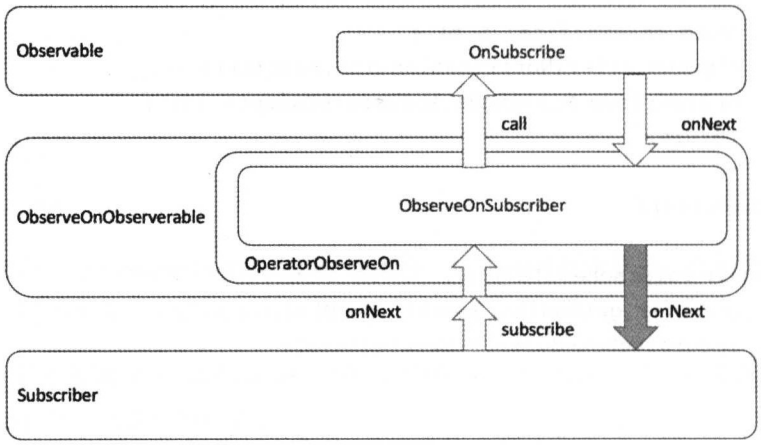

Observable
OnSubscribe
call
onNext
ObserveOnObserverable
ObserveOnSubscriber
OperatorObserveOn
onNext
subscribe
onNext
Subscriber

图 4-3-3

第 5 章

RxJava 的应用实例

在第 2 章、第 3 章中我们讲解了操作符和 Scheduler，读者看完后可能很难记住这些操作符或者某些 Scheduler 的应用。在本章中，我们将结合一些实例，让读者明白 RxJava 是如何解决实际问题的，从而加深对操作符和 Scheduler 的理解，以便更好地应用于实际工作中。

5.1 计算 π 的值

π，即为圆周率，是一个无理数，在中学几何中有大量关于 π 的题目，想必大家早就熟悉得不能再熟悉了。但是要如何来求 π 的数值呢？其实有很多方法，这里要介绍的方法是基于概率来计算的，如图 5-1-1 所示，给定一个边长为 $2r$ 的正方形和一个它的内切圆，如何求 π 的值？根据公式我们可以得到如下的等式。

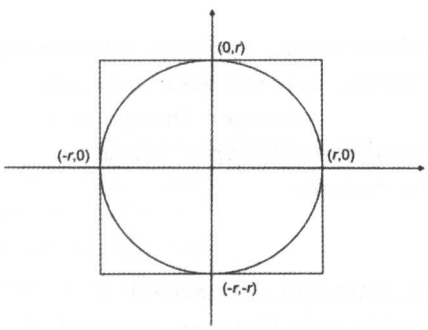


图 5-1-1

1. 正方形的面积 SA 的计算公式为 $SA=4r^2$。

2. 圆的半径是 r，则圆的面积 CA 的计算公式为 $CA=\pi r^2$

3. 知道了 SA 和 CA，则 $\pi=CA\div r^2=CA\times 4\div SA$

从上述等式可以看出，要求出 π 的值，只需要求出圆的面积和正方形面积的比值即可，而这个值可以用概率的方法来求：生成大量的在正方形范围内的随机点，落在圆内的随机点的数目除以总的随机点数就是圆和正方形面积的比值。当然这样求出来值的精确度跟随机点的数目是相关的：随机点的数目越多，精确度越高。请读者先思考一下，你会使用哪些操作符来求出 π 的值呢？

不知读者想出什么方案了吗？在这里我们将主要使用 map 和 reduce 来完成计算操作。如代码 5-1-1 所示，我们首先创建一个 Random 对象用于产生随机数，然后创建一个方法 createObservable，这个方法接收的参数 num 是要产生的随机数的个数。使用 RxJava 可以让整个计算的逻辑都显得很清晰，下面我们从上往下依次分析每个操作符的作用。

代码 5-1-1

```
private Random mRandom = new Random();
private Observable<Double> createObservable(final int num) {
    return Observable.range(0, num)
            .map(new Func1<Integer, Integer>() {
                public Integer call(Integer integer) {
                    double x = mRandom.nextDouble() * 2 - 1;
                    double y = mRandom.nextDouble() * 2 - 1;
                    return (x * x + y * y) < 1 ? 1 : 0;
                }
            })
        .reduce(new Func2<Integer, Integer, Integer>() {
                public Integer call(Integer integer,
                                    Integer integer2) {
                    int reduce = integer + integer2;
                    return reduce;
                }
            })
            .map(new Func1<Integer, Double>() {
                public Double call(Integer integer) {
                    double v = 4.0 * integer / num;
```

```
                System.out.println("V:" + v);
                return v;
            }
        })
        .subscribeOn(Schedulers.computation());
}
```

1. range：产生了 num 个数据。

2. map：将 range 产生的随机数转换为随机点的坐标，这些随机点应该确保都是在正方形的范围之内。有了随机点之后再根据随机点到原点的距离来判断随机点是否在圆的范围之内。如果在圆的范围之内，则返回 1，否则返回 0。

3. reduce：统计所有在圆的范围之内的随机点的总数。因为上面的 map 操作符会在随机点落在圆的范围之内时返回 1，所以这里只需要做简单的加法就可以。

4. map：再次用到 map 操作符，这是因为上面的 reduce 最终发送出来的数据就是 num 个数据点中落在圆内的数据点总数，所以我们可以用这个数代表圆的面积，用 num 代表正方形的面积，然后套用上面我们推导出来的公式求得 π 的估算值。

5. subscribeOn：整个计算过程没有 I/O 操作，完全是计算密集型的，所以我们使用 computation 类型的 Scheduler 来充分利用 CPU 的计算性能。

使用 createObservable 方法返回的 Observable 对象只能求出一个估算值，想要让值更精确，一方面可以加大点的数目，另一方面可以让多个 Observable 同时进行计算，最后求平均值。如代码 5-1-2 所示，我们可以根据参数来创建多个 Observable 对象，让它们同时运行，然后使用 zip 操作符将所有 Observable 求得的值汇总并取平均值。一般来讲这样求得的值会比单个 Observable 的值更精确一些。

代码 5-1-2

```
public Observable<Double> getPi(int workNum, int num) {
    ArrayList<Observable<Double>> list =
            new ArrayList<>(workNum);
    for (int i = 0; i < workNum; i++) {
        list.add(createObservable(num));
    }
    //用 zip 求得所有数据的平均值
```

```
        return Observable.zip(list, new FuncN<Double>() {
            public Double call(Object... args) {
                int len = args.length;
                double result = 0;
                for (int i = 0; i < len; i++) {
                    result += (Double) (args[i]);
                }
                return result / len;
            }
        });
    }
```

为了测试上面的程序，我们写了一个 testcase。testcase 在开发中的用处是非常大的，使用 testcase 可以测试我们写的程序是否正确，合理设计的 testcase 可以保证对程序的更改不会引入新的 bug，还可以使用 testcase 进行程序性能的测试等。RxJava 的源码中就包含了大量的 testcase，对程序的覆盖率非常高。对 testcase 不了解的读者也可以参阅 RxJava 的源码来学习 testcase 的用法。在代码 5-1-3 中的 testcase 中，我们将通过各有 1000 万个随机点的 4 个 Observable 来估算 π 的值。由于计算过程十分耗时而且是异步的，为了更好地验证结果，我们使用了 CountDownLatch 来监控计算是否完成。我们能接受的最大误差是 0.001，不知我们的程序是否能通过验证呢？

代码 5-1-3

```
final static double PiValue = 3.14159265;
@Test
public void test1() {
    final CountDownLatch latch = new CountDownLatch(1);
    PI pi = new PI();
    final double[] result = {0};
    pi.getPi(4, 10000000)
            .subscribe(new Action1<Double>() {
                public void call(Double aDouble) {
                    System.out.print(aDouble);
                    result[0] = aDouble;
                    latch.countDown();
                }
            });
```

```
        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        assertEquals(PiValue, result[0], 0.001);
    }
```

testcase 运行后的输出结果如下。我们将 4 个 Observable 计算的值和最后的平均值都打印了出来。可以看到，我们的程序求得的值在误差范围内。读者在运行这个 testcase 的时候可以尝试修改 Observable 和点的数目，看看修改后求得的结果有什么不同。

```
V:3.143012
V:3.138528
V:3.141844
V:3.144612
3.141999
```

5.2　图片的三级缓存

做过 Android 开发的读者一定对网络图片的三级缓存都很了解。为了减少流量的浪费并加快响应速度，Android 应用在加载网络图片的时候会使用三级缓存，即内存、外存（SD 卡）和网络。只要在加载网络图片时依次搜索这三级缓存，当找到图片的时候就会将图片显示在 UI 界面上并将图片保存在前面一级或两级缓存里，方便下次加载。对于图片搜索的 key 一般就是图片的 URL，图片在内存中是以 LRU（Least Recently Used）Cache 的方式存储，在外存和网络中则是以文件的形式存储。

如图 5-2-1 所示，Image1 存在于这三级网络中，所以搜索到内存的时候就可以命中并将图片返回，响应很快；而搜索 Image2 的时候，内存中并没有，所以搜索到了外存中才命中，这时需要将图片读入到内存中并返回；对于 Image3 及其他存在网络上的图片，在内存和外存中都没有，所以搜索到了网络才能命中，并且在将图片下载保存到外存和内存之后才能将图片返回。现在有很多成熟的库可以提供网络图片的下载和缓存，如 Universal-Image-Loader、Picasso、

Fresco 和 Glide 等，都很强大而且有各自的特色。虽然有了这么多轮子了，但是作为练习，在本节中我们将使用 RxJava 来实现一个 Android 系统上的网络图片三级缓存。请读者先结合三级缓存的特点思考一下怎样使用 RxJava 才能实现这个功能。

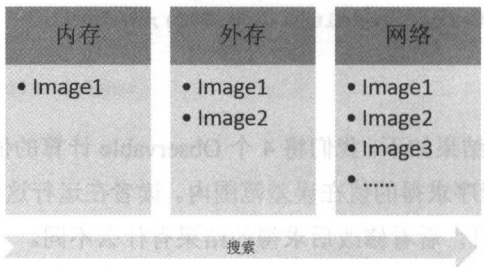


图 5-2-1

在这里我们将主要使用 concat 和 first 操作符来实现三级缓存。之所以使用 concat 而不使用 merge，是因为三级缓存对于图片的搜索顺序是有严格要求的，即必须是按照内存→外存→网络的顺序来搜索，在第 2 章我们了解到 merge 操作符是不能保证数据的发送顺序的，所以我们选择了能保证发送顺序的 concat 而不是 merge。对于搜索的图片，我们只需要找到第一个命中的缓存即可，无须继续往下搜索，所以这里我们选择了 first 操作符。总体的思路就是创建三个 Observable，分别代表内存、外存和网络，然后使用 concat 操作符将这三个 Observable 组合起来，并使用 first 操作符取第一个命中并发送出来的数据。

理解了三级缓存的原理，就让我们动手来实现吧。首先创建一个 Data 类，用来代表一张图片，并且将会作为被 Observable 发送的数据，如代码 5-2-1 所示。在 Data 类中，有成员变量 URL 作为键值用来检索；bitmap 即为我们需要加载的具体图片数据；而 isAvailable 用来标识当前的数据是否可用。我们提供了两个构造方法，既可以通过 bitmap 直接创建 Data 对象，也可以通过读文件的方式创建。

代码 5-2-1

```
public class Data {
    public Bitmap bitmap;
    public String url;
    private boolean isAvailable;
    public Data(Bitmap bitmap, String url) {
        this.bitmap = bitmap;
```

```
        this.url = url;
    }
    public Data(File f, String url) {
        if (f != null && f.exists()) {
            this.url = url;
            try {
                FileInputStream stream = new FileInputStream(f);
                bitmap = BitmapFactory.decodeStream(stream);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            }
        }
    }
    public boolean isAvailable() {
        isAvailable = url != null && bitmap != null;
        return isAvailable;
    }
}
```

接下来，我们创建一个基类 CacheObservable，这个类只有一个 getObservable 方法，接收传入的URL并根据这个URL返回一个Observable对象。我们将会创建三个继承了 CacheObservable 的类，来分别代表三级缓存的每一级，如代码 5-2-2 所示。

代码 5-2-2

```
public abstract class CacheObservable {
    public abstract Observable<Data> getObservable(String url);
}
```

做好这些准备后，我们就可以分别来实现三级缓存的各个部分了。我们将会按照由内到外的顺序来依次实现每个部分，这是因为外面缓存的实现是依赖于内部缓存的实现的。

5.2.1　内存缓存

首先让我们来创建内存缓存。图片在内存中将会存储在 LRU Cache 中，LRU Cache 有如下的特点。

- 总的存储空间有限。
- 最近访问过的或者存储的对象会放在最前面。
- 当内存不足的时候会将最不常用的对象删除。

由于应用运行时的内存十分有限，所以 LRU Cache 是一种非常理想的在内存中存储图片的数据结构，既可以有效地保存最近使用过的图片，又可以避免占用太多内存导致 OOM 错误。Android 的 supportv4 包给我们提供了一个 LruCache 类，其内部是基于 LinkedHashMap 来实现的，而 LinkedHashMap 实际是基于双向循环链表来实现的，通过双向循环链表可以很容易地满足上面提到的 LRU Cache 的 3 个特点。要实现能存储图片的缓存，我们需要继承 LruCache 类，并实现其 sizeOf 方法，如代码 5-2-3 所示。由于我们缓存的是图片，所以需要重写 LruCache 的 sizeOf 方法，将 Bitmap 大小的计算结果返回，这样 LruCache 就可以判断内存的使用量是多少，从而在内存不足的时候删除过旧的数据。

代码 5-2-3

```
public class MemoryCache<T> extends LruCache<T, Bitmap> {
    public MemoryCache(int maxSize) {
        super(maxSize);
    }
    @Override
    protected int sizeOf(T key, Bitmap b) {
        int size = 0;
        if (b != null) {
            size = b.getRowBytes() * b.getHeight();
        }
        return size;
    }
}
```

准备好了 MemoryCache，我们接下来实现 MemoryCacheObservable 类，如代码 5-2-4 所示。在这个类中，我们设置缓存的大小为 24MB，然后创建一个 Memorycache 对象来存储图片。由于我们需要往内存缓存中放入刚刚用到的图片，所以我们还需要提供一个 putData 方法，将新的图片数据插入到 MemoryCache 中。由于我们在基类中定义了虚函数 getObservable，所以 MemoryCacheObservable 类也需要实现这个方法。在这个方法中，我们使用 just 操作符创建一

个 Observable 对象并返回。这个 Observable 会根据传入的 URL 从 MemoryCache 中查询图片数据，并创建一个 Data 数据发送出来。可能有的读者会有疑问，当 MemoryCache 中没有对应的图片时，我们查询得到的 Bitmap 对象肯定为空，这样我们创建的 Data 会安全而且有必要吗？请不要担心，我们在 Data 中提供了 isAvailable 方法，在使用 Data 的时候会首先调用该方法来确保数据的有效性。前文中已经提过，我们将会使用 concat 操作符来组合所有的 Observable 对象，而 concat 操作符会严格地保证只有前一个 Observable 发送完数据后，后一个 Observable 才能开始发送数据，所以当 Bitmap 为空的时候我们也发送数据，这样这个 Observable 才不会阻塞后面的 Observable。

代码 5-2-4

```
public class MemoryCacheObservable extends CacheObservable {
    //缓存大小为 24MB
    public static final int CACHE_SIZE = (24 * 1024 * 1024);
    MemoryCache<String> mCache = new MemoryCache<>(CACHE_SIZE);
    @Override
    public Observable<Data> getObservable(String url) {
        return Observable.just(url)
                .map(new Func1<String, Data>() {
                    @Override
                    public Data call(String s) {
                        Logger.i("search in memory");
                        return new Data(mCache.get(url), url);
                    }
                });
    }
    public void putData(Data data) {
        mCache.put(data.url, data.bitmap);
    }
}
```

5.2.2　外存缓存

由于外存中的数据会以文件的形式保存，所以我们无须像内存那样创建一个 LruCache 对象来存储数据，而需要做好文件的读写操作。如代码 5-2-5 所示，我们创建一个 DiskCacheObservable

类，用一个 Context 对象来获取文件的存储路径，一个 File 对象来代表存储在外存中的图片文件。在 getObservable 方法中，我们同样使用 just 操作符创建一个 Observable 对象并返回，这个 Observable 对象发送的数据将会是我们通过文件作为参数来创建的 Data 对象。

代码 5-2-5

```
public class DiskCacheObservable extends CacheObservable {
    Context mContext;
    File mCacheFile;
    public DiskCacheObservable(Context mContext) {
        this.mContext = mContext;
        mCacheFile = mContext.getCacheDir();
    }
    @Override
    public Observable<Data> getObservable(String url) {
        return Observable.just(url)
                .map(new Func1<String, Data>() {
                    @Override
                    public Data call(String s) {
                        Logger.i("read file from disk");
                        File f = getFile(url);
                        Data data = new Data(f, url);
                        return data;
                    }
                })
                .subscribeOn(Schedulers.io());
    }
    private File getFile(String url) {
        url = url.replaceAll(File.separator, "-");
        return new File(mCacheFile, url);
    }
    /**
     * 将下载的图片保存到外存中
     *
     * @param data 要保存的图片数据
     */
    public void putData(Data data) {
    ...
    }
}
```

外存缓存需要将网络缓存下载的图片保存起来，所以我们同样需要提供 putData 方法。如代码 5-2-6 所示，在接收到网络缓存发送的 Data 数据后，我们通过 just 操作符来创建 Observable 对象将 Data 发送出来，然后在 map 操作符将图片压缩并存储到文件中。由于文件的读写操作可能会发生异常，所以我们将捕捉到的异常通过 Exceptions.propagate(e) 封装成一个 RuntimeException 并抛出来，这样异常发生时 RxJava 就会捕捉到该异常并通过 onError 方法将异常发送给订阅者——我们在任何操作符的内部碰到异常时都可以这样处理。另外文件的读写操作都是耗时操作，所以 getObservable 方法返回的 Observable 对象和这里的 Observable 对象都会被订阅到 io 类型的 Scheduler 上。细心的读者可能会发现，我们在最后只是调用了 subscribe 方法，并没有将一个具体的 Subscriber 订阅到这个 Observable 上，所以如果真的有异常发生也根本没有 Subscriber 通过 onError 方法来接收并处理异常。如果存储文件发生异常，其实并不会影响整个缓存的使用，只是外存这一级会失效而已，所以对异常的处理就不是那么重要了。感兴趣的读者可以自己来实现这一部分异常的处理。

代码 5-2-6

```
public void putData(Data data) {
    Observable.just(data)
            .map(new Func1<Data, Data>() {
                @Override
                public Data call(Data data) {
                    File f = getFile(data.url);
                    OutputStream out = null;
                    try {
                        out = new FileOutputStream(f);
                        Bitmap.CompressFormat format;
                        if (data.url.endsWith("png")
                                || data.url.endsWith("PNG")) {
                            format = Bitmap.CompressFormat.PNG;
                        } else {
                            format = Bitmap.CompressFormat.JPEG;
                        }
                        data.bitmap.compress(format, 100, out);
                        out.flush();
                        out.close();
                    } catch (IOException e) {
                        throw Exceptions.propagate(e);
                    } finally {
```

```
                if (out != null) {
                    try {
                        out.close();
                    } catch (IOException e) {
                        throw Exceptions.propagate(e);
                    }
                }
            }
            return data;
        }
    }).subscribeOn(Schedulers.io()).subscribe();
}
```

5.2.3 网络缓存

同前两级缓存一样，网络缓存 NetCacheObservable 同样继承自 CacheObservable 类。如代码 5-2-7 所示，在 getObservable 方法中，我们还是使用 just 方法来创建 Observable 对象，然后在 map 方法中通过 URLConnection 从网络上读取图片数据并根据读取到的图片数据创建 Bitmap 对象。有了 Bitmap 对象，我们就可以创建 Data 对象并发送出来了。由于网络操作最耗时，所以我们也会将 Observable 对象订阅到 io 类型的 Scheduler 上。网络缓存是最外面一层缓存了，所以不需要向前两级缓存一样提供 putData 方法。

代码 5-2-7

```
public class NetCacheObservable extends CacheObservable {
    @Override
    public Observable<Data> getObservable(String url) {
        return Observable.just(url)
                .map(new Func1<String, Data>() {
                    @Override
                    public Data call(String url) {
                        Bitmap bitmap = null;
                        InputStream inputStream = null;
                        Logger.i("get img on net:" + url);
                        try {
                            URLConnection con = new URL(url)
                                    .openConnection();
```

```
                    inputStream = con.getInputStream();
                    bitmap = BitmapFactory
                            .decodeStream(inputStream);
                } catch (IOException e) {
                    throw Exceptions.propagate(e);
                } finally {
                    if (inputStream != null) {
                        try {
                            inputStream.close();
                        } catch (IOException e) {
                            throw Exceptions.propagate(e);
                        }
                    }
                }
                return new Data(bitmap, url);
            }
        }).subscribeOn(Schedulers.io());
    }
}
```

5.2.4　缓存管理

三级缓存都创建好了，可以拿来用了吧？还不行，还记得我们在内存缓存和外存缓存的实现时都预留了 putData 方法吗？因为当外层的缓存拿到图片数据并组装成 Data 对象发送出来的时候，我们还需要调用内层缓存的 putData 方法将数据保存到内层的缓存里，这样当重复使用图片时，三级缓存才能真正发挥作用。所以接下来我们还需要实现缓存的管理功能，如代码 5-2-8 所示，我们创建了一个 Sources 类来管理所有的缓存。可以看到在 Sources 类内部有一个 Context 对象是传给 MemoryCacheObservable 用来创建文件的存储目录用的；有三级缓存的三个对象，我们将会对其进行管理操作；还有对外提供的 memory、disk 和 network 三个方法，会返回各种对应类型的 Observable 对象；最后我们还有一个 logSource 方法，会返回一个 Transformer 对象，在这个 Transformer 对象内部，我们使用 doOnNext 方法来输出 Data 的状态。使用 compose 操作符可以将这个 Transformer 对象应用到每个 Observable 对象上，我们就可以通过日志来监控每个 Observable 对象的方法。如果读者忘记了 compose 的使用方法，可以回到 2.10.2 节重新复习一下。

代码 5-2-8

```
public class Sources {
    Context mContext;
    MemoryCacheObservable mMemoryCacheOvservable;
    DiskCacheObservable mDiskCacheObservable;
    NetCacheObservable mNetCacheObservable;
    public Sources(Context mContext) {
        this.mContext = mContext;
        mMemoryCacheOvservable = new MemoryCacheObservable();
        mDiskCacheObservable = new DiskCacheObservable(mContext);
        mNetCacheObservable = new NetCacheObservable();
    }
    public Observable<Data> memory(String url) {
        …
    }
    public Observable<Data> disk(String url) {
        …
}
    public Observable<Data> network(String url) {
        …
}
Transformer<Data, Data> logSource(final String source) {
    return dataObservable -> dataObservable.doOnNext(data -> {
        if (data != null && data.isAvailable()) {
            Logger.i(source + " has the data!");
        } else {
            Logger.i(source + " not has the data!");
        }
    });
}
```

对于 memory 方法的实现，我们无须做什么管理操作，只要使用 compose 操作符加上 log 就可以直接返回，如代码 5-2-9 所示。

代码 5-2-9

```
public Observable<Data> memory(String url) {
    return mMemoryCacheOvservable.getObservable(url)
            .compose(logSource("MEMORY"));
}
```

对于 disk 方法的实现，我们就需要做一些额外的操作了。如代码 5-2-10 所示，我们首先通过 compose 操作符加上日志信息，然后对于内存缓存发送出来的数据使用 filter 操作符做一下过滤操作，只保留有效的数据。

代码 5-2-10

```
public Observable<Data> disk(String url) {
    return mDiskCacheObservable.getObservable(url)
            .compose(logSource("DISK"))
            .filter(new Func1<Data, Boolean>() {
                @Override
                public Boolean call(Data data) {
                    return data.isAvailable();
                }
            })
            //将图片保存到外存中
            .doOnNext(new Action1<Data>() {
                @Override
                public void call(Data data) {
                    mMemoryCacheOvservable.putData(data);
                }
            });
}
```

同 disk 方法一样，也需要对 network 方法做额外的操作。如代码 5-2-11 所示，我们首先通过 compose 操作符加上日志信息，然后使用 filter 操作符过滤出有效的数据，然后在 doOnNext 操作符中将数据保存到外存和内存中。

代码 5-2-11

```
public Observable<Data> network(String url) {
    return mNetCacheObservable.getObservable(url)
            .compose(logSource("NET"))
            .filter(new Func1<Data, Boolean>() {
                @Override
                public Boolean call(Data data) {
                    return data.isAvailable();
                }
            })
```

```
            .doOnNext(new Action1<Data>() {
                @Override
                public void call(Data data) {
                    //将图片保存到外存和内存中
                    mMemoryCacheOvservable.putData(data);
                    mDiskCacheObservable.putData(data);
                }
            });
    }
```

5.2.5 封装

万事俱备，只差最后的封装操作并提供一个对外的接口了。如代码 5-2-12 所示，我们对外提供了一个 loadImage 接口，用户将用于显示图片的 ImageView 和图片的 URL 作为参数传递进来，我们会返回一个 Observable 对象给用户，这样用户就可以自由地操作这个 Observable 对象了。在最终得到图片时我们会自动帮用户将图片加载到 ImageView 上，而用户可以通过 Subscriber 得到这张图片的数据。

代码 5-2-12

```
public class RxImageLoader {
    static Sources sources;
    public static void init(Context mContext) {
        sources = new Sources(mContext);
    }
    private static final Map<Integer, String> cacheKeysMap
            = Collections.synchronizedMap(new HashMap<>());
    /**
     *
     * 将制订 URL 地址的图片加载到 ImageView 中
     *
     * @param img 用来显示图片的 ImageView
     * @param url 要显示的图片 url
     * @return 一个 Observable,可以同 RxJava 结合起来使用
     */
    public static Observable<Data> loadImage(ImageView img,
                                             String url) {
```

```
        if (img != null) {
            cacheKeysMap.put(img.hashCode(), url);
        }
        Observable<Data> source = Observable.concat(
                sources.memory(url),
                sources.disk(url),
                sources.network(url))
                .first(new Func1<Data, Boolean>() {
                    @Override
                    public Boolean call(Data data) {
                        return data != null
                                && data.isAvailable()
                                && url.equals(data.url);
                    }
                })
.observeOn(AndroidSchedulers.mainThread());
        return source.doOnNext(new Action1<Data>() {
            @Override
            public void call(Data data) {
                int hashcode = img.hashCode();
                String cachedUrl = cacheKeysMap.get(hashcode);
                if (img != null && url.equals(cachedUrl)) {
                    img.setImageBitmap(data.bitmap);
                }
            }
        });
    }
}
```

loadImage 的具体实现方法就是上文提到的使用 concat 操作符将内存、外存和网络对应的 Observable 对象结合起来，并使用 first 操作符来过滤出第一个符合条件的有效数据，最后在 doOnNext 操作符里将图片加载到 ImageView 上。由于这里涉及了 UI 操作，所以在最后使用 observeOn(AndroidSchedulers.mainThread())将更新 UI 的操作切换回主线程。此外在 ListView 中有 View 的重用机制，所以为了防止滑动时加载图片出现错位现象，我们会使用一个线程安全的 Map 来保存 ImageView 和 URL 的对应关系。

5.2.6 运行测试

RxImageLoader 封装好了，现在写一个 demo 运行测试一下。我们将会创建一个 List 并插入很多图片的 URL（图片都来自网络），然后通过一个 ListView 将图片都展示出来。当然这些图片的加载都是通过我们刚刚封装好的 RxImageLoader 来实现的。如代码 5-2-13 所示，首先调用 init 方法进行初始化，然后创建一个继承自 BaseAdapter 的 RxAdapter 类，在加载图片时使用 RxImageLoader.loadImage 接口，最后将 RxAdapter 设置到 ListView 上。

代码 5-2-13

```
RxImageLoader.init(getApplicationContext());

class RxAdapter extends BaseAdapter {
    @Override
    public int getCount() {
        return contents.size();
    }
    @Override
    public String getItem(int i) {
        return contents.get(i);
    }
    @Override
    public long getItemId(int i) {
        return i;
    }
    @Override
    public View getView(int i, View view, ViewGroup viewGroup) {
        ViewHolder holder;
        if (view == null) {
            holder = new ViewHolder();
            view = View.inflate(MainActivity.this,
                    R.layout.activity_main, null);
            holder.img = (ImageView) view.findViewById(R.id.img);
            view.setTag(holder);
        } else {
            holder = (ViewHolder) view.getTag();
        }
        holder.img.setImageResource(R.mipmap.ic_launcher);
```

```
        RxImageLoader
                .loadImage(holder.img, getItem(i))
                .subscribe();
        return view;
    }
}

getListView().setAdapter(new RxAdapter());
```

运行 demo，结果如图 5-2-2 所示。

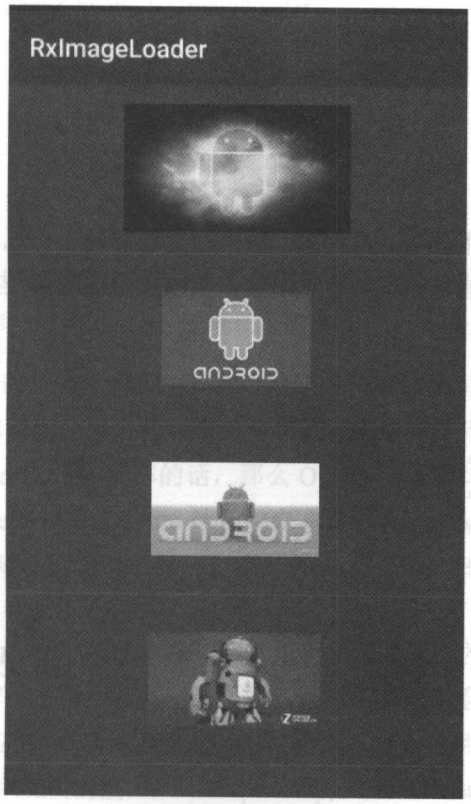


图 5-2-2

第一次运行时输出的 log 如下。第一次加载在内存和外存中肯定是没有数据的，所以这两级缓存里的数据肯定是无效的，最后在网络缓存里找到数据。

```
MEMORY do not has the data!
MEMORY do not has the data!
DISK do not has the data!
DISK do not has the data!
MEMORY do not has the data!
DISK do not has the data!
MEMORY do not has the data!
DISK do not has the data!
MEMORY do not has the data!
DISK do not has the data!
NET has the data!
NET has the data!
NET has the data!
NET has the data!
NET has the data!
```

接下来将 ListView 向下滑动，直到第一屏的所有图片都不可见，然后再往回滑动回到起始的状态，这时 log 的输出信息如下。可见，这一次图片的加载都是从内存中加载的，根本就没有到外存和网络上去查询。

```
MEMORY has the data!
MEMORY has the data!
MEMORY has the data!
MEMORY has the data!
MEMORY has the data!
```

退出程序，再次运行的结果如下。可以看到，由于上次启动 App 的时候图片已经被缓存到外存中，所以这一次图片会从外存中加载，无须再一次从网络上下载了。

```
MEMORY do not has the data!
MEMORY do not has the data!
MEMORY do not has the data!
MEMORY do not has the data!
DISK has the data!
MEMORY do not has the data!
DISK has the data!
```

```
DISK has the data!
DISK has the data!
DISK has the data!
```

通过实际测试可知我们用 RxJava 实现的这个三级缓存运行良好，而且使用起来十分方便，在功能上可能没有本节开头时所提到的那几个知名库强大，但是足以胜任一般图片的加载了。最重要的是，通过实现这个三级缓存，我们对于用到的操作符如 concat、first、map、compose 和 doOnNext 等会有更加深刻的理解，也了解了如何从操作符的内部抛出异常。该项目的完整代码可以参考 https://github.com/Chaoba/RxImageloader。

5.3　结合 Retrofit 和 OkHttp 访问网络

现在基本上所有的 Android 应用都需要通过网络和服务器进行交互，所以在 Github 上已经有了很多著名的网络访问库，其中 Square 公司开发的 Retrofit 和 OkHttp 属于比较强大的库。

OkHttp 是一个网络访问库，通过 OkHttp 可以发出各种网络请求；而 Retrofit 则是一种网络访问库的封装框架，其内部可以使用多种网络访问库进行网络的访问，而且可以和 RxJava 无缝结合起来使用。如果将 Retrofit 比作汽车的话，那么 OkHttp 就是汽车的发动机，而汽车是可以用多种类型的发动机的。Retrofit 和 OkHttp 结合起来非常适合 Android 开发者们开发网络应用。在本节中，我们将使用 RxJava 结合 Retrofit 和 OkHttp 开发一个炉石传说卡片的浏览应用。

5.3.1　卡片类的定义

要实现一个网络访问应用，毫无疑问我们需要一个服务器来提供访问接口。幸运的是现在网上有很多网站可以给我们提供免费的接口。Mashape（https://market.mashape.com/）上就有各种各样的免费接口可供访问，不过这是一个国外的网站，国内访问可能会有点慢。在 Mashape 上我们选择了炉石的接口，我们的应用将会访问这里的接口。Mashape 总共提供了 11 个 Get 请求接口，以满足我们按照不同的规则查询各种卡片和卡背的需求，在本书中我们不会将这 11 个接口都讲述一遍，而是会选择一个比较有代表性的接口展开下面的内容。

由于卡片和背卡都是一种卡牌，所以首先我们将其公共的部分提取出来创建一个 BaseCard 类，如代码 5-3-1 所示。在 BaseCard 里有 4 个基本属性，分别是：

- cardId：卡片的 ID，每个卡片的 ID 都是唯一的。
- name：卡片的名字。
- img：卡片的图片链接。
- locale：语言的地区。

代码 5-3-1

```
public class BaseCard {
    public String cardId;
    public String name;
    public String img;
    public String locale;
}
```

然后我们来定义卡片类，如代码 5-3-2 所示。卡片类有更多的属性，这里就不一一解释了。要注意的一点就是卡片类的信息会被用于在不同的界面之间传递，所以我们的这个类需要实现 Parcelable 接口。

代码 5-3-2

```
public class Card extends BaseCard implements Parcelable{
    public String cardSet;
    public String faction;
    public String rarity;
    public String type;
    public String health;
    public String imgGold;
    public String collectible;
    public String playerClass;
    public String cost;
    public String attack;
    public String text;
    public String flavor;
    public String artist;
```

```
    public String race;
    public String howToGet;
    public String howToGetGold;
    public List<BaseCard> mechanics;
    public String menuType;
    protected Card(Parcel in) {
    …
        }
    public static final Creator<Card> CREATOR = new Creator<Card>() {
        @Override
        public Card createFromParcel(Parcel in) {
            return new Card(in);
        }
        @Override
        public Card[] newArray(int size) {
            return new Card[size];
        }
    };
    @Override
    public int describeContents() {
        return 0;
    }
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        …
    }
}
```

5.3.2　配置 OkHttp

OkHttp 有一个很强大的功能，就是可以给 Request 和 Response 都加上 Interceptor。所谓的 Interceptor 就是在统一的出口和入口的地方对 Request 和 Response 的头部信息做增加、删除和修改的操作。我们将会创建两个 Interceptor，分别对 Request 和 Response 的头部增加一些必要的信息。

Mashape 要求用户注册一个账号，然后创建一个 key，并在发送的请求头部加上这个 key 的信息，这样才能正确返回请求的内容。所以我们先创建一个 MashapeKeyInterceptor 类，让其

实现 Interceptor 接口，如代码 5-3-3 所示。我们在 Request 的头部加上了"X-Mashape-Key"的信息，这样就能通过 Mashape 服务器的验证了，不要忘记将"yourKey"替换你自己生产的 key。此外由于我们需要使用的数据是 Json 格式，所以我们还会额外添加一个"application/json"的信息，这样服务器就会给我们返回 Json 格式的数据。有了这个 Interceptor，并且设置好以后，所有发出的请求都会带有我们在这里配置的两个头部信息。

代码 5-3-3

```
public class MashapeKeyInterceptor implements Interceptor {
    @Override
    public Response intercept(Chain chain) throws IOException {
        Request request = chain.request();
        Request.Builder builder = request.newBuilder();
        Request newRequest = builder
                .addHeader("X-Mashape-Key", "yourKey")
                .addHeader("Accept", "application/json")
                .build();
        return chain.proceed(newRequest);
    }
}
```

一般来讲，这套炉石卡片的接口不会频繁更新，所以我们完全可以将数据缓存起来，而无须每次都去接口请求，这样应用的响应会更快，用户体验也就更好。OkHttp 可以根据 Response 头部的"Cache-Control"的信息来自动地帮我们完成缓存，而 Mashape 默认返回的 Response 头部并没有包含缓存的信息，所以 OkHttp 不会自动地帮我们做缓存。如果想让 OkHttp 帮我们做缓存就需要修改 Response 头部加上"Cache-Control"的信息，所以我们还需要新建一个 CacheInterceptor。如代码 5-3-4 所示，在 CacheInterceptor 当中，我们给 Response 的头部加上了缓存 30 天的信息。

代码 5-3-4

```
public class CacheInterceptor implements Interceptor {
    @Override
    public Response intercept(Chain chain) throws IOException {
        Request request = chain.request();
        Response response = chain.proceed(request);
        Response response1 = response.newBuilder()
```

```
                .removeHeader("Pragma")
                .removeHeader("Cache-Control")
                //缓存 30 天
                .header("Cache-Control",
                        "max-age=" + 3600 * 24 * 30)
                .build();
        return response1;
    }
}
```

虽然我们实现了 CacheInterceptor，但是 OkHttp 并不知道将缓存的信息保存在哪，也不知道缓存的大小是多少，所以我们还需要创建一个 Cache 实例并将其设置到 OkHttpClient 里面，如代码 5-3-5 所示。

代码 5-3-5

```
public Cache provideCache() {
    return new Cache(mContext.getCacheDir(), 10240*1024);
}
```

准备好了两个 Interceptor 和 Cache，我们就可以创建 OkHttpClient 了。如代码 5-3-6 所示，我们创建一个 OkHttpClient 对象，并将自己实现的两个 Interceptor 和 Cache 设置进去。不仅如此，为了更好地调试程序，我们还加入了 HttpLoggingInterceptor，它属于 OkHttp 的一个子模块，这样在 Debug 模型下就可以将我们发出的 Request 和接收的 Response 信息打印出来。

代码 5-3-6

```
public OkHttpClient provideOkHttpClient(Cache cache) {
    OkHttpClient okHttpClient = new OkHttpClient();
    HttpLoggingInterceptor httpLoggingInterceptor =
            new HttpLoggingInterceptor();
    httpLoggingInterceptor
            .setLevel(BuildConfig.DEBUG ? BODY : NONE);
    OkHttpClient newClient = okHttpClient.newBuilder()
            .addInterceptor(httpLoggingInterceptor)
            .addInterceptor(new MashapeKeyInterceptor())
            .addNetworkInterceptor(new CacheInterceptor())
            .cache(cache)
```

```
            .connectTimeout(30, TimeUnit.SECONDS)
            .readTimeout(30, TimeUnit.SECONDS)
            .build();
    return newClient;
}
```

至此 OkHttpClient 就准备好了，相当于汽车的发动机装配好了。接下来我们把 Retrofit 这辆汽车也装配好，然后将发动机装入汽车，汽车就可以开动了。

5.3.3 配置 Retrofit

我们要访问的所有接口都返回 Json 格式的字符串，所以我们将会使用 Google 的 Gson 库来解析 Json 字符串。但是有些接口会返回 JsonArray，而 Gson 默认是不能解析 JsonArray 的，所以为了解析 JsonArray，我们首先要实现 JsonDeserializer 接口。如代码 5-3-7 所示，我们实现了一个 CardsDeserializer 类，并在创建 Gson 对象时将其注册进去，这样当遇到 JsonArray 时，Gson 就会帮我们将它解析成 List<Card>类型的数据。

代码 5-3-7

```
public class CardsDeserializer implements
        JsonDeserializer<List<Card>> {
    @Override
    public List<Card> deserialize(JsonElement je, Type typeOfT,
                                  JsonDeserializationContext
                                          context)
            throws JsonParseException {
        Type listType = new TypeToken<List<Card>>() {
        }.getType();
        List<Card> cards = null;
        try {
            cards = new Gson().fromJson(je, listType);
        } catch (JsonSyntaxException e) {
            CLogger.e(e);
        }
        return cards;
    }
}
```

```
Gson gson = new GsonBuilder()
    .registerTypeAdapter(new TypeToken<List<Card>>() {
            }.getType(),
            new CardsDeserializer())
    .create();
```

接下来我们根据 Mashape 上对接口的定义来实现一个 IApi 接口。Retrofit 的接口定义是用注解来实现的，如代码 5-3-8 所示，我们定义了两个 Get 接口，第一个接口 getInfo 请求的路径是 info；而第二个接口请求的路径是 cards，还要加上 locale 的值。除了这里用到的注解，Retrofit 还有 Post、Put、Multipart 和 Feild 等注解，在这里我们就不展开了，读者可以查阅 Retrofit 的官网来了解其他注解的使用。回到我们定义的这两个接口，可以看到其返回的对象是我们很熟悉的 Observable 对象，而且 Retrofit 已经帮我们将 Json 字符串转化为我们之前定义好的对象了，所以 Observable 就可以把这些对象作为数据发送出来。这时我们就可以通过订阅来拿到请求的数据了，当然还可以使用操作符对数据做一些转化、过滤操作等。

代码 5-3-8

```
public interface IApi {
    @GET("info")
    Observable<Info> getInfo();
    @GET("cards")
    Observable<List<Card>>
    getAllCards(@Query("locale") String locale);
    …
}
```

现在 Retrofit 需要的所有零件我们都准备好了。如代码 5-3-9 所示，我们创建一个 Builder 对象，首先设置 BASE_URL，这样 IApi 里定义的所有接口都会在这个 BASE_URL 的基础上被组装成最终的请求接口；第二步添加 RxJavaCallAdapterFactory，这样 Retrofit 才能让 IApi 里的接口返回 Observable 类型的对象；第三步添加上根据我们之前定义的 Gson 对象创建的 GsonConverterFactory，Retrofit 将根据我们添加的规则来解析服务器返回的 Json 字符串；第四步会将之前准备好的 OkHttpClient 设置为 Retrofit 的 client，Retrofit 就会使用这个 OkHttpClient 来访问网络。有了 Retrofit 对象，再调用其 create 方法，将我们之前定义好的 IApi 接口类传进去，我们就能得到一个实现了 IApi 接口的对象，从而调用 IApi 接口里的方法来访问网络。

代码 5-3-9

```
private static final String BASE_URL =
        "https://omgvamp-hearthstone-v1.p.mashape.com";
public IApi getCardsApi(OkHttpClient client) {
    RxJavaCallAdapterFactory adapter =
            RxJavaCallAdapterFactory.create();
    GsonConverterFactory gsonConverter =
            GsonConverterFactory.create(gson);
    Retrofit CardsApiAdapter = new Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addCallAdapterFactory(adapter)
            .addConverterFactory(gsonConverter)
            .client(client)
            .build();
    return CardsApiAdapter.create(IApi.class);
}
```

得到了实现 IApi 接口的对象，我们就可以自由调用之前定义好的方法发出请求了。如代码 5-3-10 所示，我们调用 getInfo 接口返回一个发送数据为 Info 的 Observable 方法，由于网络请求比较耗时，所以我们在 io 类型的 Scheduler 上进行订阅并在主线程上进行监听。当请求的数据返回时，我们在 onNext 方法里收到了一个 Info 对象，然后根据这个 Info 对象里的信息来更新 UI。

代码 5-3-10

```
Observable<Info> observable = mIApi.getInfo();
observable.subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new SelfDefineSubscriber<Info>() {
            @Override
            public void onNext(Info info) {
                mInfo = info;
                mView.onUpdate(mInfo);
                mView.hideProgress();
            }
        });
```

至此，我们已经将 Retrofit、OkHttp 和 RxJava 串起来了，这里只是列举出了一些关键的代码，整个项目的源码可到 https://github.com/Chaoba/HearthstoneCards 下载。在项目中我们还用到了 Dagger2，但那不是本书中要讨论的内容了，对 Dagger2 感兴趣的读者也可以参阅源码里的使用方式。

5.4　使用 RxLifecycle 避免内存泄漏

Android 中的 Activity 和 Fragment 有着自己的生命周期，所以如果当 Activity 或者 Fragment 结束时其内部还有正在工作着的订阅，而你又忘了进行反订阅的话，那么这部分的资源是不会释放的，这就会导致内存的泄漏。或许你会想，那我在 onDestory 方法中反订阅一下，问题不就解决了吗？这确实是一种办法，但是如果 Activity 和 Fragment 多了，你就不得不在每个 Activity 或者 Fragment 的 onDestory 方法里面反订阅一遍，十分麻烦，而且还很容易遗忘。要应对这种情况我们可以引入 RxLifecycle。

RxLifecycle（https://github.com/trello/RxLifecycle/tree/1.x）是一个开源的工具库，可以根据 Activity 和 Fragment 的生命周期事件来结束一次订阅，这样使用 RxLifecycle 我们就可以在创建一个 Observable 时指定它在哪个生命周期里被反订阅，而无须在所有使用过 RxJava 的 Activity 和 Fragment 的 on XXX 方法中进行无聊的反订阅操作。

5.4.1　修改 demo 工程

在第 2 章讲解 RxJava 中的操作符时，我们给出的示例代码都是在 Android 工程中写的，并分散在一个个单独的 Activity 中，这样有很多 Acticity 在退出时会出现上文所说的内存泄漏问题。下面我们就通过 RxLifecycle 来解决这个问题。

首先需要在工程的 gradle 里面加入如下配置，来导入对 RxLifecycle 的依赖。

```
compile 'com.trello:rxlifecycle-android:1.0'
compile 'com.trello:rxlifecycle-components:1.0'
```

由于我们在 demo 工程中提前准备了一个 BaseActivity，而所有的 Activity 都继承自

BaseActivity，所以我们只需要修改 BaseActivity，让其继承 RxLifecycle 提供的 RxActivity，这样所有的 Activity 就都会继承 RxActivity。

```
public class BaseActivity extends RxActivity
```

然后我们将所有的 Observable 都通过.compose(bindToLifecycle())转化一下，这样 RxLifecycle 就可以在 Acticity 退出之前帮我们自动反订阅所有的 Observable 了。或许有的读者会想，这么多 Observable，每个都添加.compose(bindToLifecycle())好麻烦，这里告诉大家一个使用 AndroidStudio 的小技巧：选择菜单 Edit→Find→Replace in Path，进入全局替换界面，如图 5-4-1 所示。

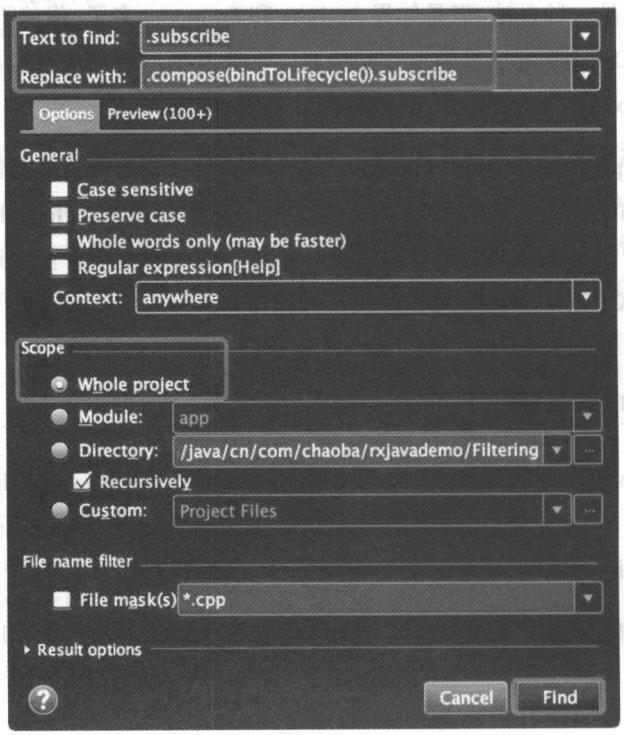


图 5-4-1

在 Text to find 后面的输入框中填入我们所要搜索并替换掉的字符串.subscribe，然后在 Replace with 后面的输入框里填入想要替换为的字符串.compose(bindToLifecycle()).subscribe，最

后在下面的 Scope 中选中 Whole project 并点击按钮 Find。如果想要进一步控制查找范围，还可以选中下面的 File mask(s)并在其后面的输入框中输入*.java 就可以只搜索 Java 源文件了。点击 Find 之后会弹出一个要求确认的对话框，如图 5-4-2 所示，选中 All Files 按钮就可以将所有订阅的地方替换，这时我们就可以放心地退出任何一个 Activity 而无须担心内存泄漏的问题了。

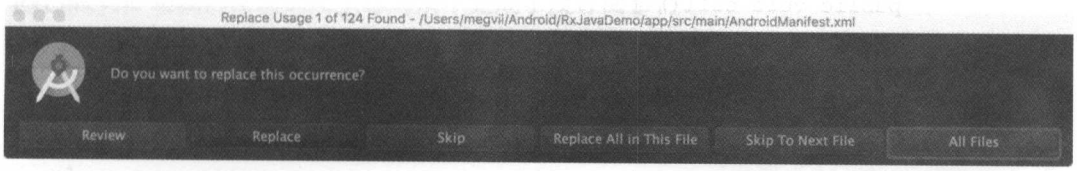


图 5-4-2

那么 RxLifecycle 到底是在什么时候帮忙进行反订阅的呢？由于我们使用的是 compose(bindToLifecycle())，所以反订阅的时机取决于订阅的时机，基本和 Activity 的生命周期相对应，具体的对应关系如下。

- 在 onCreate 中订阅对应在 onDestroy 中反订阅。
- 在 onStart 中订阅对应在 onStop 中反订阅。
- 在 onResume 中订阅对应在 onPause 中反订阅。
- 在 onPause 中订阅对应在 onStop 中反订阅。
- 在 onStop 中订阅对应在 onDestroy 中反订阅。
- 在 onDestroy 中订阅对应抛出异常。

5.4.2　绑定其他生命周期

在 5.4.1 节中我们了解了 RxLifecycle 会根据订阅的时机来自动帮我们在对应的生命周期里进行反订阅操作。那如果我们想更灵活地控制反订阅的时机，该怎么办呢？RxLifecycle 提供了 bindUntilEvent 方法可以让我们自由地控制反订阅的时机，并且对 Acticity 和 Fragment 的各个生命周期都定义了 enum，方便我们直接使用。在代码 5-4-1 中，我们使用 interval 操作符创建了两个 Observable，每隔 500 毫秒发送出一个数据，然后使用 RxLifecycle.bindUntilEvent 将其反订阅的时机绑定到 onPause 和 onDestory 中，而 onPause 和 onDestory 方法也被我们重写来输出日志。

代码 5-4-1

```
Observable.interval(500, TimeUnit.MILLISECONDS)
        . compose(RxLifecycle.bindUntilEvent(lifecycle(), ActivityEvent.PAUSE))
        .doOnUnsubscribe(new Action0() {
            @Override
            public void call() {
                log("pause onUnsubscribe");
            }
        })
        .subscribe(new Action1<Long>() {
            @Override
            public void call(Long aLong) {
                log(aLong);
            }
        });

Observable.interval(500, TimeUnit.MILLISECONDS)
        .compose(RxLifecycle.bindUntilEvent(lifecycle(),
                ActivityEvent.DESTROY))
        .doOnUnsubscribe(new Action0() {
            @Override
            public void call() {
                log("destroy onUnsubscribe");
            }
        })
        .subscribe(new Action1<Long>() {
            @Override
            public void call(Long aLong) {
                log(aLong);
            }
        });
@Override
protected void onPause() {
    super.onPause();
    log("onPause");
}
@Override
protected void onDestroy() {
    super.onDestroy();
```

```
        log("onDestroy");
    }
```

运行程序启动 Acitivity 让 Observable 开始发送数据后再退出，输出结果如下，可以看到 RxLifecycle 帮我们在指定的生命周期中完成了反订阅事件。

```
0
1
2
0
1
pause onUnsubscribe
onPause
destroy onUnsubscribe
onDestroy
```

5.5　使用 RxBinding 绑定各种 View 事件

Android 有各种各样的 View 来完成不同功能，如 TextView、ListView、RecyclerView、EditText 和 Button 等。这些 View 都可以绑定一些公有的事件，如点击事件，也可以绑定一些各自特有的事件，如 EditText 的文本改变事件和 ListView 的 item 点击事件等。这么多的事件如果以匿名内部类的形式添加到代码中的话，一方面各种不同的 Listener 看起来乱，另一方面有些 Listener 包含了很多我们可能根本用不到的方法，但是我们却不得不保留这些方法的空实现。这时我们就可以使用 RxBinding 来绑定 View 的各种事件。RxBinding（https://github.com/JakeWharton/RxBinding）是 JakeWharton 开发的开源库，用来绑定 Android 中的各种 View 并将其转变为 Observable，转变后的 Observable 可以发送各种与 View 相关的事件从而可以让我们用 RxJava 的操作符来对这些事件做转化、过滤等操作。

可以通过在工程的 gradle 里面加入如下的配置来导入对 RxLifecycle 的依赖。

```
compile 'com.jakewharton.rxbinding:rxbinding:1.0.1'
```

这只是对 Android 基础库的支持，RxBinding 还有各种扩展库来支持 Android 的各种扩展库。

- 支持 support-v4 扩展包：

```
compile 'com.jakewharton.rxbinding:rxbinding-support-v4:1.0.1'
```

- 支持 appcompat-v7 扩展包：

```
compile 'com.jakewharton.rxbinding:rxbinding-appcompat-v7:1.0.1'
```

- 支持 design 扩展包：

```
compile 'com.jakewharton.rxbinding:rxbinding-design:1.0.1'
```

- 支持 recyclerview-v7 扩展包：

```
compile 'com.jakewharton.rxbinding:rxbinding-recyclerview-v7:1.0.1'
```

下面我们将分别通过绑定 Android 基础库和扩展库里的控件作为实例来了解一下 RxBinding 的使用。

5.5.1 绑定点击事件

对于 Button 来讲，基本都是对其添加一个 OnClickListener 来监听点击事件的，但是如果迅速地多次点击 Button 就会多次触发其点击事件，在很多情况下这不是我们想要的。那要怎么处理呢？或许你可以如代码 5-5-1 那样创建一个成员变量 lastTime 来存储上次点击的时间，然后每次点击时都更新一下 lastTime 的值，并计算距离上次点击的间隔时间，如果间隔时间小于 200 毫秒则返回，否则往下处理点击事件。这样实现一看就有很多问题，首先针对每个 Button 都要创建一个成员变量来记录时间，其次计算间隔时间的代码需要在每个点击事件中都实现一遍，

这简直是不能忍受的。

代码 5-5-1

```
mRButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        long currentTime = System.currentTimeMillis();
        long duration = currentTime - lastTime;
        lastTime = currentTime;
        if (duration < 200) {
            return;
        }
        log("right click" + duration);
    }
});
```

这时你可能想起了在第 3 章中有操作符可以对数据进行过滤，那我们能不能结合 RxBinding 来将过多的点击事件过滤掉呢？答案是肯定的。在代码 5-5-2 中，我们是使用 RxBinding 里的方法 RxView.clicks(mLButton)将 mLButton 的点击事件通过一个 Observable 发送出来，然后我们就可以使用 RxJava 中的操作符 throttleWithTimeout 将过多的点击事件过滤掉了。需要注意的是，throttleWithTimeout 的过滤行为与我们上面给出来的例子并不完全相同，throttleWithTimeout 会将事件间隔小于 200 毫秒的事件都过滤掉，最终发送出来的是最后一个事件。所以当你快速多次点击时，不会收到任何点击事件；只有当你速度慢下来或者停止点击时，才会有一个点击事件被发送出来，而前面的都会被过滤掉。

代码 5-5-2

```
RxView.clicks(mLButton)
        .throttleWithTimeout(200, TimeUnit.MILLISECONDS)
        .subscribe(new Action1<Void>() {
            @Override
            public void call(Void aVoid) {
                log("click");
            }
        });
```

这样即使有很多的 Button 需要过滤快速点击事件，我们也可以很完美地解决。下面我们继续来看一下 TextWatcher。

5.5.2 绑定 TextWatcher

当我们想要监听 TextView 的内容变化时，我们可以对其添加一个 TextWatcher，如代码 5-5-3 所示。TextWatcher 内有三个方法，而我们一般会用 onTextChaned 方法来获取 TextView 文字的变化。其余的方法即使我们不用也必须将其实现，哪怕仅仅是一个空的实现。这会增加很多无用的代码，多少会给我们带来一些困扰。

代码 5-5-3

```
mResultView.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s,int start, int count, int after)
{

    }
    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count)
{
        // 做一些工作
    }
    @Override
    public void afterTextChanged(Editable s) {
    }
});
```

RxBinding 可以很轻松地帮我们解决这个问题，在代码 5-5-4 中，我们使用 RxTextView.textChanges(mResultView)将 TextView 的文字变化通过 Observable 发送出来，然后通过 map 得到字符串的长度，并使用 filter 来过滤掉字符串长度小于等于 10 的事件，这样当文字的长度大于 10 的时候，我们就可以在 Subscriber 里接收到通知。使用 RxBinding 不仅避免了冗余的代码，还可以利用 RxJava 的链式操作来对这些文字做其他后续的操作。

代码 5-5-4

```
RxTextView.textChanges(mResultView)
        .map(new Func1<CharSequence, Integer>() {
            @Override
            public Integer call(CharSequence charSequence) {
                return charSequence.length();
            }
        })
        .filter(new Func1<Integer, Boolean>() {
            @Override
            public Boolean call(Integer integer) {
                return integer > 10;
            }
        })
        .subscribe(new Action1<Integer>() {
            @Override
            public void call(Integer integer) {
                Log.e(TAG, "Text changed to much times: " + integer);
            }
        });
```

但是在实际工作中，我们可能就是需要 beforeTextChanged 或者 afterTextChanged 的回调信息，那该怎么办？放心，在 RxTextView 中是包含这两个方法所对应的 Rx 方法的，分别是 RxTextView.afterTextChangeEvents 和 RxTextView.afterTextChangeEvents 方法，此外还有 RxTextView.color、RxTextView.hint 等方法基本可以满足我们的各种需求。

5.5.3　绑定 OnPageChangeListener

ViewPager 在 Android 开发中应用得十分广泛，通过将一个 PageAdapter 绑定到 ViewPager，开发者可以轻松地实现左右滑动切换显示界面的功能。在代码 5-5-5 中，我们通过一个 ViewPager 来展示显示数字的子界面，我们设定的子界面个数为 Integer.MAX_VALUE，但我们使用了一个小技巧，只创建了 5 个 View，通过 position 对 5 取余的方式就可以让 ViewPager 包含 Integer.MAX_VALUE 个子界面，这样就不用每次都创建新的 View，可以获得较高的性能。

代码 5-5-5

```
ArrayList<View> viewList = new ArrayList(5);
LayoutInflater li = getLayoutInflater();
for (int i = 0; i < 5; i++) {
    viewList.add(li.inflate(R.layout.viewpager_item, null));
}
ViewPager viewPager = (ViewPager) findViewById(R.id.view_pager);
viewPager.setAdapter(new PagerAdapter() {
    @Override
    public Object instantiateItem(ViewGroup container, int position) {
        View child = viewList.get(position % 5);
        container.addView(child);
        TextView textView = (TextView) child.findViewById(R.id.item_text);
        textView.setText(String.valueOf(position));
        return child;
    }
    @Override
    public void destroyItem(ViewGroup container, int position, Object object) {
        container.removeView(viewList.get(position % 5));
    }
    @Override
    public int getCount() {
        return Integer.MAX_VALUE;
    }
    @Override
    public boolean isViewFromObject(View view, Object object) {
        return view == object;
    }
});
```

当我们需要随时知道当前子界面的 position 时，我们需要给 ViewPager 添加一个 OnPageChangeListener。但是同 TextWatcher 一样，OnPageChangeListener 包含了 3 个方法，而我们这时只需要用 onPageSelected 方法，如代码 5-5-6 所示，虽然我们只用了一个方法，但是却不得不添加其他两个方法的空实现。

代码 5-5-6

```
viewPager.addOnPageChangeListener(new ViewPager.OnPageChangeListener() {
    @Override
```

```
    public void onPageScrolled(int position, float positionOffset, int
          positionOffsetPixels) {
    }
    @Override
    public void onPageSelected(int position) {
        log("onPageSelected:" + position);
    }
    @Override
    public void onPageScrollStateChanged(int state) {
    }
});
```

RxBinding 同样可以解决这个问题，在代码 5-5-7 中，我们通过 RxViewPager. pageSelections (viewPager)得到了一个发送当前显示的子界面的 position 的 Observable，可以使我们只关注自己想要的数据。此外，我们还可以通过 RxViewPager 的 currentItem 和 pageScrollStateChanges 方法来分别设定 ViewPager 显示子界面的 position 和监听其滑动事件。

代码 5-5-7

```
RxViewPager.pageSelections(viewPager)
        .subscribe(new Action1<Integer>() {
            @Override
            public void call(Integer integer) {
                log("RxViewPager onPageSelected:" + integer);
            }
        });

RxViewPager.currentItem(viewPager).call(Integer.MAX_VALUE / 2);
RxViewPager.pageScrollStateChanges(viewPager)
        .subscribe(new Action1<Integer>() {
            @Override
            public void call(Integer integer) {
                log("pageScrollStateChanges:" + integer);
            }
        });
```

在本章中，我们通过实例展示了如何使用 RxJava 来解决具体问题，并且简单介绍了网上的一些大牛们基于 RxJava 创建的开源库。

第 6 章

RxJava 2 的改进

RxJava 2 于 2016 年 10 月正式发布，并且完全符合 Reactive-Streams 的标准。Reactive-Streams 是一种在 Java 虚拟机上进行异步数据流编程的协议，在本书里将不会就 Reactive-Streams 的具体内容做展开说明，感兴趣的读者可以到 Github 上 Reactive-Streams 的代码仓库里查看详细的内容。

伴随着 RxJava 2 的发布，很多依赖于 RxJava 的库如 RxAndroid 和 Retrofit 等也都做了相应的升级，可以使用 RxJava 2 的最新功能。相比 RxJava 1，RxJava 2 具有更好的性能、更低的开销和更多的功能，所以我们可以在自己的工作中尽量使用 RxJava 2。接下来让我们来了解一下 RxJava 2 都做了哪些改进。

本章中的实例代码都是基于 RxJava 2 实现的。

6.1 Observable 和 Flowable

在 RxJava 2 中不仅仅有 Observable 可以接受订阅并发送数据，还添加了一个新的类型 Flowable。主要是因为在 RxJava 1 中，Observable 默认并没有实现 backpressure 机制。backpressure 是用来做什么的呢？当 Observable 发送数据的速度大于 Subscriber 消费的速度时，就会造成大量未处理的数据积累，占用大量系统资源，这时就需要用 backpressure 来减缓 Observable 发送数据的速度。在这种情况下如果没有 backpressure 就会很容易出现 MissingBackpressureException 或者 OutOfMemoryError 的异常。而 Folwable 则在内部默认实现了 backpressure 机制，所以就不

用担心上文中所提到的问题。那我们在什么情况下使用 Observable，又在什么情况下使用 Flowable 呢？

优先使用 Observable 的情况如下。

- 要发送的数据总量不超过 1000 个。这种规模的数据一般很少会将系统内存全部消耗完，所以也就不会出现 OutOfMemoryError 异常。

- 处理 GUI 上的输入事件时，如鼠标移动或者是触摸事件等。这种情况下使用 sample 或者 debounce 操作符做一下限流，无须 backpressure 也不会有什么问题。

- 数据流是同步的。也就是说 Observable 发送数据和 Subscriber 消耗数据是在同一个线程里进行的，这种情况下只有 Subscriber 消费掉了一个数据，Observable 才能产生下一个数据，所以不存在大量数据无法消费造成的累积问题。

因为 Observable 没有 backpressure 机制，所以一般情况下其性能要比 Flowable 高。所以在以上几种情况下我们要优先考虑使用 Observable。

优先使用 Flowable 的情况如下。

- 要发送的数据量超过 10000 个的情况。这种情况下一般需要控制一下数据源发送数据的速度。

- 从文件读取数据时。读取文件操作一般是异步的，所以当文件的数据量较大而我们每次只能处理较少的数据时，就需要使用 backpessure 了。

- 从数据库读取数据时。这种情况和上面的从文件读取数据很像。

- 使用网络写入或者读取数据流时。这种情况下一般都需要控制每次写入或者读取的数量，也适合使用 backpressure 来控制。

以上几种情况容易造成大量未处理的数据累积，或者需要控制每次操作的数据总量，所以需要 backpressure 机制来控制数据源发送数据的速度，我们优先选择使用 Flowable。

下面我们分别使用 range 操作符创建一个 Flowable 对象和一个 Obervalbe 对象，如代码 6-1-1 所示，分别发送 10000 个数据，然后使用 map 操作符延迟一秒来模拟一个处理较慢的 Consumer，并统计它们的总运行时间。

代码 6-1-1

```
long time = System.currentTimeMillis();
Flowable.range(1, 10000)
        .subscribeOn(Schedulers.io())
        .map(new Function<Integer, Integer>() {
            @Override
            public Integer apply(@NonNull Integer i) throws Exception {
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return i;
            }
        })
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                log(integer);
                if (integer == 10000) {
                    log(System.currentTimeMillis() - time);
                }
            }
        });
long time = System.currentTimeMillis();
Observable.range(1, 10000)
        .subscribeOn(Schedulers.io())
        .map(new Function<Integer, Integer>() {
            @Override
            public Integer apply(@NonNull Integer i) throws Exception {
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return i;
            }
        })
        .subscribe(new Consumer<Integer>() {
            @Override
```

```
            public void accept(Integer integer) throws Exception {
                log(integer);
                if (integer == 10000) {
                    log(System.currentTimeMillis() - time);
                }
            }
        });
```

运行后 log 输出了 Flowable 运行时间，为 13807 毫秒，而 Observable 的运行时间为 13718 毫秒。这是由于 Flowable 内有 backpressure 机制，所以其性能比 Observable 稍差了一些。

6.2　null 的使用

不知读者在使用 RxJava 1 的时候是否注意到，在 RxJava 1 中 null 是可以作为一种数据被发送出来，并可以被 Subscriber 接收的。但是在 RxJava 2 中，null 不再被看作是一种可以发送的数据了。我们看一下代码 6-2-1，首先我们用 just 操作符创建一个发送数据为 1 的 Observable，然后使用 map 操作符返回一个 null 类型的数据后进行订阅；接下来我们直接创建一个发送 null 的 Observable 进行订阅，看看会发生什么。

代码 6-2-1

```
Observable.just(1)
        .map(new Function<Integer, Object>() {
            @Override
            public Object apply(@NonNull Integer integer) throws Exception {
                return null;
            }
        })
        .subscribe(new Observer<Object>() {
            @Override
            public void onSubscribe(@NonNull Disposable d) {
            }
            @Override
            public void onNext(@NonNull Object o) {
            }
```

```
            @Override
            public void onError(@NonNull Throwable e) {
                log(e.getMessage());
            }
            @Override
            public void onComplete() {
            }
        });
Observable.just(null).subscribe(new Observer<Object>() {
    @Override
    public void onSubscribe(@NonNull Disposable d) {
    }
    @Override
    public void onNext(@NonNull Object o) {
    }
    @Override
    public void onError(@NonNull Throwable e) {
        log(e.getMessage());
    }
    @Override
    public void onComplete() {
    }
});
```

运行后会首先输出如下的 log，说明在 map 操作符里会直接捕获到 null 的异常，然后将其包装成 Throwable 后，通过 Observer 的 onError 方法将错误抛了出来。

```
The mapper function returned a null value.
```

之后继续往下运行，则程序会直接崩溃，给出的崩溃栈里的信息如下：

```
java.lang.NullPointerException: The item is null
```

所以无论是新创建的 Observable 还是通过别的操作符进行过转化的 Observable，都不应该再将 null 作为数据发送出来了。

6.3　Single 和 Completable

我们对 Single 类型其实已经很了解了，它可以并且只可以发送出一个数据或者一个错误事件。在 RxJava 2 中 Single 被完全重新设计了，与它对应的订阅者 SingleSubscriber 也改为了 SingleObserver。如代码 6-3-1 所示，SingleObserver 增加了一个 onSubscribe 方法，在订阅后可以得到一个 Disposable 对象，这个 Disposable 对象可以用来取消订阅。

代码 6-3-1

```
interface SingleObserver<T> {
    void onSubscribe(Disposable d);
    void onSuccess(T value);
    void onError(Throwable error);
}
```

下面我们使用 just 操作符创建两个 Single 对象，如代码 6-3-2 所示，对于第一个我们直接进行订阅，而对于第二个则延迟 1 秒再发送数据，同时我们会在 onSubscribe 方法中持有其 Disposable 对象并调用这个 Disposable 对象的 dispose 方法来取消订阅。让我们运行程序来看看会输出什么结果。

代码 6-3-2

```
Single.just(1)
        .subscribe(new SingleObserver<Integer>() {
            @Override
            public void onSubscribe(@NonNull Disposable d) {
                log("onSubscribe");
            }
            @Override
            public void onSuccess(@NonNull Integer integer) {
                log("onSuccess:" + integer);
            }
            @Override
            public void onError(@NonNull Throwable e) {
            }
        });
Disposable disposable;
```

```
Single.just(1)
        .delay(1, TimeUnit.SECONDS)
        .subscribe(new SingleObserver<Integer>() {
            @Override
            public void onSubscribe(@NonNull Disposable d) {
                log("onSubscribe");
                disposable = d;
            }
            @Override
            public void onSuccess(@NonNull Integer integer) {
                log("onSuccess:" + integer);
            }
            @Override
            public void onError(@NonNull Throwable e) {
            }
        });
disposable.dispose();
```

运行后的输出结果如下。可以看到第一个 Single 对象在调用其 SingleObserver 的 onSubscribe 方法后通过 onSuccess 方法将数据 1 发送了出来。而对于第二个 Single 对象，因为我们在其发送数据前就取消了订阅，所以并没有任何数据被发送出来。

```
onSubscribe
onSuccess:1
onSubscribe
start dispose
```

而 Completable 在使用上和 RxJava 1 中相比并没有什么变化，只是其订阅者也添加了 onSubscribe 接口。Completable 的定义如代码 6-3-3 所示。

代码 6-3-3

```
interface CompletableObserver<T> {
    void onSubscribe(Disposable d);
    void onComplete();
    void onError(Throwable error);
}
```

6.4 Maybe

Maybe 是 RxJava 2 新引入的一种类型，在发送数据的时候，至多只能发送一个数据；当不发送数据的时候就只能发送出一个错误事件或者一个结束事件。Maybe 可以被认为是 Single 和 Completable 的结合体。下面我们创建两个 Maybe 对象，如代码 6-4-1 所示，第一个 Maybe 对象发送一个数据；而第二个 Maybe 对象在发送出一个数据后再发送一个 onComplete 事件，看一下运行结果。

代码 6-4-1

```
Maybe.just("1")
        .subscribe(new MaybeObserver<String>() {
            @Override
            public void onSubscribe(@NonNull Disposable d) {
                log("onSubscribe");
            }
            @Override
            public void onSuccess(@NonNull String s) {
                log("onSuccess:" + s);
            }
            @Override
            public void onError(@NonNull Throwable e) {
                log("onError");
            }
            @Override
            public void onComplete() {
                log("onComplete");
            }
        });
Maybe.create(new MaybeOnSubscribe<String>() {
    @Override
    public void subscribe(@NonNull MaybeEmitter<String> e) throws Exception {
        if(!e.isDisposed()){
            e.onSuccess("1");
            e.onComplete();
        }
    }
}).subscribe(new MaybeObserver<String>() {
```

```
        @Override
        public void onSubscribe(@NonNull Disposable d) {
            log("onSubscribe");
        }
        @Override
        public void onSuccess(@NonNull String s) {
            log("onSuccess:" + s);
        }
        @Override
        public void onError(@NonNull Throwable e) {
            log("onError:" + e.getMessage());
        }
        @Override
        public void onComplete() {
            log("onComplete");
        }
    });
```

运行后的输出结果如下。可以看到两个 Maybe 对象都是发送出了一个数据，而并没有发送出任何 onError 或者 onComplete 事件。所以当我们确定数据源只能发送出零个或一个数据的时候，Maybe 是一种比较好的选择。

```
onSubscribe
onSuccess:1
onSubscribe
onSuccess:1
```

6.5 Subscriber

相比 RxJava 1，RxaJva 2 中的 Subscriber 都多了一个 onSubscribe 方法，会在订阅到 Flowable 或者 Observable 时被调用，另外 onCompleted 方法改为了 onComplete 方法，去掉了一个字母 d。为了避免命名混淆，在 RxJava 2 中提供了三种订阅者来代替 Subscriber，分别是 DefaultSubscriber、ResourceSubscriber 和 DisposableSubscriber。

下面我们来分别了解一下这三个订阅者的特点。

6.5.1　DefaultSubscriber

DefaultSubscriber 实现了 FlowableSubscriber 接口，并且将 onSubscribe 实现为 final，所以使用 DefaultSubscriber 的话是无法重写 onSubscribe 方法的。在其 onSubscribe 方法的实现里，会默认帮我们请求 Long.MAX_VALUE 个数据。DefaultSubscribe 还提供了一个 protected 方法 cancel，可以在其 on XXX 方法内调用来取消订阅。

下面我们就新建一个 DefaultSubscriber 对象，如代码 6-5-1 所示，在其 onNext 方法里判断接收的数据是否为 3，如果为 3，就通过 cancel 方法取消订阅。然后将其订阅到一个使用 interval 操作符创建的 Flowable 对象上，该 Flowable 对象每隔一秒发送一个数据。

代码 6-5-1

```
DefaultSubscriber<Long> subscriber = new DefaultSubscriber<Long>() {
    @Override
    public void onNext(Long t) {
        log("onNext:" + t);
        if (t == 3) {
            cancel();
        }
    }
    @Override
    public void onError(Throwable t) {
        log("onError:" + t.getMessage());
    }
    @Override
    public void onComplete() {
        log("onComplete");
    }
};
Flowable.interval(1, TimeUnit.SECONDS).subscribe(subscriber);
```

运行后的输出结果如下，DefaultSubscriber 只接收到了数字 3 以前的数据，然后就会取消订阅。

```
onNext:0
onNext:1
onNext:2
onNext:3
```

6.5.2 ResourceSubscriber

ResourceSubscriber 实现了 FlowableSubscriber 和 Disposable 接口。其特点就是有一个 add 方法可以将其他的 Disposable 作为一个资源添加进来，然后在 onError 和 onComplete 中需要调用 dispose 方法，这样会使添加过的 Disposable 对象都取消订阅。

在代码 6-5-2 中，我们首先创建了一个 ResourceSubscriber 对象 subscriber，然后将一个订阅到 Flowable 对象所返回的 Disposable 作为资源通过 add 方法添加进来，最后将 subscriber 订阅到一个 Flowable 对象。

代码 6-5-2

```
ResourceSubscriber<Long> subscriber = new ResourceSubscriber<Long>() {
    @Override
    public void onNext(Long t) {
        log("onNext:" + t);
    }
    @Override
    public void onError(Throwable t) {
        log("onError:" + t.getMessage());
        dispose();
    }
    @Override
    public void onComplete() {
        log("onComplete");
        dispose();
    }
};
    Disposable disposable = Flowable.interval(1, TimeUnit.SECONDS).subscribe(new
Consumer<Long>() {
    @Override
    public void accept(Long aLong) throws Exception {
```

```
            log("added resource:" + aLong);
        }
    });
    subscriber.add(disposable);
    Flowable.interval(1, TimeUnit.SECONDS).subscribe(subscriber);
    //几秒钟后
    subscriber.dispose();
```

运行代码几秒后，我们调用 subscriber 的 dispose 方法，输出的结果如下。可以看到两个订阅者都输出了三个数据。在调用了 subscriber.dispose()之后，所有的订阅者都取消了订阅，不再接收数据。

```
added resource:0
onNext:0
added resource:1
onNext:1
added resource:2
onNext:2
```

6.5.3　DisposableSubscriber

DisposableSubscriber 实现了 FlowableSubscriber 和 Disposable 接口。除了多了 Disposable 接口以外，其他的实现都跟 DefaultSubscriber 一样。如代码 6-5-3 所示，我们创建了一个 DisposableSubscriber 对象 subscriber，然后将 subscriber 订阅到了一个 Flowable 对象上。我们可以在内部的 on XXX 方法里使用 cancel 方法取消订阅，也可以在外部调用 subscriber.dispose()方法来取消订阅。

代码 6-5-3

```
DisposableSubscriber<Long> subscriber = new DisposableSubscriber<Long>() {
    @Override
    public void onNext(Long t) {
        log("onNext:" + t);
        if (t == 3) {
            cancel();
```

```
        }
    }
    @Override
    public void onError(Throwable t) {
        log("onError:" + t.getMessage());
    }
    @Override
    public void onComplete() {
        log("onComplete");
    }
};

Flowable.interval(1, TimeUnit.SECONDS).subscribe(subscriber);
…
subscriber.dispose();
```

订阅后输出的结果如下，在输出了数据 3 之后，由于我们在 onNext 方法内调用了 cancel 方法，所以取消了订阅。

```
onNext:0
onNext:1
onNext:2
onNext:3
```

6.6 Action 和 Function

在 RxJava 1 中，Action 和 Function 是两种非常重要的对象集合，每个集合内都包含了 9 个对象，可以接收 0~9 个甚至更多个参数。它们的命名也都跟参数有关联，如 Action0~Action9、Action*N*、Func0~Func9、Func*N* 等。当我们使用 map 操作符的时候，就需要传入一个 Function1 对象；当我们订阅到一个 Observable 时，如果只想接收 Observable 发送出的数据而不关心错误和结束信息，就可以直接用一个 Action1 对象来进行订阅。在 RxJava 1 的源码内部也大量地使用了 Action 和 Function 集合。

在 RxJava 2 中，对 Action 和 Function 集合都按照 Java 8 的命名风格做了改变，对 Action

集合具体的改变如下。

- 在操作符内使用的 Action0 都被替换成 io.reactivex.functions.Action。
- 在 Scheduler 内使用的 Action0 都被替换成 java.lang.Runnable。
- Action1 被重命名为 Consumer。
- Action2 被重命名为 BiConsumer。
- Action3~Action9 以及 Action*N* 被 Consumer<Object[]>代替。

在前面的实例代码中，我们已经多次使用过 Consumer，在下面的代码中我们来使用一下 BiConsumer。如代码 6-6-1 所示，我们使用 just 和 error 操作符创建了两个 Single 对象，分别发送一个数据 1 和一个错误事件，然后分别使用 BiConsumer 进行订阅。

代码 6-6-1

```
Single.just(1).subscribe(new BiConsumer<Integer, Throwable>() {
    @Override
    public void accept(Integer v, Throwable e) throws Exception {
        log(v);
    }
});
Single.error(new RuntimeException("test error"))
        .subscribe(new BiConsumer<Object, Throwable>() {
            @Override
            public void accept(Object v, Throwable e) throws Exception {
                log(e);
            }
        });
```

BiConsumer 接收两个参数，第一个参数接收发送来的数据，第二个参数接收错误事件。订阅后的输出结果如下，可以看到第一个 BiConsumer 接收到了数据，而第二个 BiConsumer 接收到了错误事件。

```
1
java.lang.RuntimeException: test error
```

Function 集合的更改同上面 Action 的更改很像，具体更改如下。

- Func1 改为 Function。
- Func2 改为 BiFunction。
- Func3 ~ Func9 改为 Function3 ~ Function9。
- Func*N* 被 Function<Object[], R>所替代。

在前面的代码中，我们已经用过了 Function，在代码 6-6-2 中，我们将使用一下 BiFunction。我们首先使用 just 操作符创建一个发送 1、2、3、4 的 Observable 对象，然后使用 reduce 操作符接收一个 BiFunction，进行数据的转化，最后通过 Consumer 接收 reduce 汇总后的数据。订阅后会输出所有数据之和，即 10。

代码 6-6-2

```
Observable.just(1, 2, 3, 4)
        .reduce(new BiFunction<Integer, Integer, Integer>() {
            @Override
            public Integer apply(Integer t1, Integer t2) {
                return t1 + t2;
            }
        })
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                log(integer);
            }
        });
```

6.7 错误处理

在 RxJava 2 中,所有的 Throwable 错误都不能在内部被过滤掉，需要抛出给外层调用方。但是如果错误是发生在订阅生命周期结束后或者是订阅被取消后，就不能直接通过 onError 方法将错误抛出了，这时错误就会被转发给 RxJavaPlugins 的 onError 方法，在 onError 方法内部会做以下三件事情。

- RxJavaPlugins 内部有一个默认为空的 errorHandler 成员变量，可以通过 setErrorHandler(Consumer<? super Throwable> handler)方法为其设置一个 handler。如果已经设置了 errorHandler，就会把 Throwable 发送到 errorHandler 里。
- 如 errorHandler 为空，则将 Throwable 的调用栈信息打印出来。
- 如 errorHandler 为空，则将 Throwable 抛给当前线程上的 UncaughtExceptionHandler。

为了更好地区分出不同种类的异常，并了解异常发生的原因，RxJava 还定义了以下几种异常包装类。

- OnErrorNotImplementedException：表示用户在订阅时没有添加错误处理。
- ProtocolViolationException：表示在操作符内部有 bug。
- UndeliverableException：因为订阅生命周期的原因而无法发送出去的异常会被包装成这种异常包装类，并且会完好地保存调用栈信息来帮助用户查找错误源。

在代码 6-7-1 中，我们通过 Single.error 来抛出一个异常，然后在没有处理错误信息的情况下直接进行订阅；然后 RxJavaPlugins.setErrorHandler 方法设置一个 hanlder 后再次通过 Single.error 来抛出一个异常。

代码 6-7-1

```
Single.error(new RuntimeException("test error")).subscribe();

RxJavaPlugins.setErrorHandler(new Consumer<Throwable>() {
    @Override
    public void accept(Throwable throwable) throws Exception {
        log(throwable.getClass().getName() + " - " + throwable.getMessage());
    }
});
Single.error(new RuntimeException("test error")).subscribe();
```

运行程序后我们会发现，在未设置 errorHandler 的情况下会直接导致程序崩溃；而在设置了 errorHandler 之后会输出如下的结果，可见这一次异常被包装成了 OnErrorNotImplementedException，并发送到了我们之前设置的 errorHandler 里面，同时程序也不会崩溃了。

```
io.reactivex.exceptions.OnErrorNotImplementedException - test error
```

6.8 Scheduler

同 RxJava 1 相比，RxJava 2 依然保留了 computation、io、newThread 和 trampoline 类型的 Scheduler，但是去掉了 immediate 类型，在需要使用 immediate 的地方都可以使用 trampoline 来代替。另外 RxJava 2 还增加了一个 single 类型的 Scheduler，这种类型的 Scheduler 在内部会开一个共享的单线程进行工作，非常适合对序列有严格要求的场景。在代码 6-8-1 中，我们创建两个 flowable，它们都会发送 1、2、3 三个数据，然后再通过 flatMap 操作符将每个数据转化为发送三个新数据的 Flowable。不同之处在于前者都是运行在 io 类型的 Scheduler 上，而后者则是运行在 single 类型的 Scheduler 上。

代码 6-8-1

```
Flowable.just(1, 2, 3)
        .flatMap(new Function<Integer, Publisher<String>>() {
            @Override
            public Publisher<String> apply(@NonNull Integer integer) throws
                    Exception {
                return Flowable.just(integer + "-1", integer + "-2", integer + "-3")
                        .subscribeOn(Schedulers.io());
            }
        })
        .subscribe(new Consumer<String>() {
            @Override
            public void accept(String s) throws Exception {
                log(s);
            }
        });
Flowable.just(1, 2, 3)
        .flatMap(new Function<Integer, Publisher<String>>() {
            @Override
            public Publisher<String> apply(@NonNull Integer integer) throws
                    Exception {
                return Flowable.just(integer + "-1", integer + "-2", integer + "-3")
```

```
                .subscribeOn(Schedulers.single());
        }
    })
    .subscribe(new Consumer<String>() {
        @Override
        public void accept(String s) throws Exception {
            log(s);
        }
    });
```

运行后输出的结果如下。可以看到，因为前者运行在 io 类型的 Scheduler 上，所以输出的数据顺序是打乱的；而后者因为是运行在 single 类型的 Scheduler 上，所有的 Flowable 使用一个共享的单线程，所以输出的数据是有序的。

```
1-1
2-1
2-2
2-3
3-1
3-2
3-3
1-2
1-3
1-1
1-2
1-3
2-1
2-2
2-3
3-1
3-2
3-3
```

在本章中，我们了解了 RxJava 2 的改进之处。虽然 RxJava 2 做了不少改进，但是其使用起来和 RxJava 1 还是很类似的，所以如果学会了 RxJava 1，掌握了响应式编程的思想再来学习 RxJava 2 就会觉得很简单。鉴于 RxJava 1 在 2018 年 3 月 31 日后不再维护，所以想要体验 RxJava

新的功能并及时得到 bug 修复的同学可以逐步向 RxJava 2 上迁移了。此外，RxJava 3 也在构思的过程中，应该很快就会开始开发，有志于对开源做出贡献的同学可以积极参与 RxJava 3 的开发，和国际上的大牛一起工作，相信一定会受益匪浅。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

为什要使用响应式编程?

响应式编程提高了代码的抽象层级，我们只需要关注与业务逻辑相关的事件，而不必去纠结里面的细节。不仅如此，响应式编程的链式调用还可以消除嵌套回调，所以使用响应式编程写出来的代码往往会更加简明易懂。响应式编程非常适合以下几种情况：

◎ 鼠标的点击、移动事件，键盘的输入事件，移动设备上的各种触摸和手势事件

◎ 当用户的位置改变时，其移动设备上的GPS信号和陀螺仪信号

◎ 各种耗时的操作，如读取硬盘内容以及从网络请求数据，这些操作一般都是异步的

◎ 涉及数据的转化、组合、过滤等操作的场景

学习RxJava最难的地方在哪儿?

最难的是编程思想的转变，学习者需要以响应式编程的方式来思考，也就是说要放弃熟知的命令式编程的思维习惯。要理解RxJava的思想，比如可以将Observable看作工厂的原材料生产机器，发送出来的数据即为原材料，整个链式操作可以视为原材料经过一条流水线，每个操作符为流水线上的一个车间，每个车间都会对原材料做一定的加工，最终的Subscriber可以视为最终消费者，会接收加工后的成品。

RxJava是一个非常优秀的开源库，其清晰的流式操作和便捷的线程切换为Java和Android开发者提供了有力的帮助。尽管网络上有很多介绍RxJava的文章，开发者也可以很容易地查找到相关的学习资料，但是由于RxJava入门比较困难，导致很多初学者浅尝辄止，放弃了继续学习和使用RxJava的机会，十分可惜。希望本书能成为一个向导，带领读者走进RxJava的世界，享受编程的乐趣。

上架建议：移动开发

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：许　艳
责任编辑：张春雨
封面设计：侯士卿

ISBN 978-7-121-33640-9

定价：49.00元